TURING 图灵程序设计丛书

[PACKT] PUBLISHING



[波兰] Rafał Kuć　Marek Rogoziński 著　蔡建斌 译

Elasticsearch 服务器开发（第2版）

Elasticsearch Server *Second Edition*

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

作 者 简 介

Rafał Kuć

solr.pl网站联合创始人，现为Sematext集团顾问和软件工程师，专注于Apache Lucene、Solr、Elasticsearch和Hadoop等开源技术。Rafał拥有超过12年的多领域软件经验，其中既包括银行软件又包括电子商务产品。Rafał也是*Apache Solr 3.1 Cookbook*等技术图书的作者，并且一直是Lucene Eurocon、Berlin Buzzwords、ApacheCon和Lucene Revolution等会议的演讲嘉宾。

Marek Rogoziński

solr.pl网站联合创始人，拥有10年以上的软件架构师和顾问从业经验，专门研究基于Solr和Elasticsearch等开源搜索引擎的解决方案，以及Hadoop、HBase和Twitter Storm等用于大数据分析的软件。

译 者 简 介



蔡建斌

敏捷践行者，擅长Scrum/XP/Kanban等敏捷实践，现在英孚教育全球研发中心任Technical Lead，除了50%时间写代码以外，业务需求分析、前后端架构设计、性能调优、自动化测试、流程改进、发布运维、代码评审……无所不为，只为开发出更好的软件。爱好围棋，弈城4段5段之间跳跃。目标：工作上有所不为；爱好上添加一项健身。Email：caijianbin93@126.com 。

图灵程序设计丛书



[波兰] Rafał Kuć　Marek Rogoziński　著　　蔡建斌　译

Elasticsearch 服务器开发（第2版）

Elasticsearch Server *Second Edition*

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

Elasticsearch服务器开发 : 第2版 / (波) 库赛,
(波) 罗格辛斯基著 ; 蔡建斌译. -- 北京 : 人民邮电
出版社, 2015.3
(图灵程序设计丛书)
ISBN 978-7-115-38032-6

Ⅰ. ①E… Ⅱ. ①库… ②罗… ③蔡… Ⅲ. ①互联网
络—情报检索 Ⅳ. ①G354.4

中国版本图书馆CIP数据核字(2014)第306232号

内 容 提 要

本书这一版针对 Elasticsearch 的最新版本更新了内容，增加了第 1 版中遗漏的重要内容。本书首先对 Elasticsearch 作一般性介绍，其中包括如何启动和运行 Elasticsearch、Elasticsearch 的基本概念，以及如何以最基本的方式索引和搜索数据。接下来，本书讨论了 Querydsl 查询语言，通过它可以创建复杂的查询并过滤返回的结果。此外，本书还展示了如何使用切面技术（faceting）基于查询结果来计算汇总数据，如何使用新引进的聚合框架，如何使用 Elasticsearch 的空间搜索和预搜索。最后，这本书将向你展示 Elasticsearch 的管理 API，如分片安置控制和集群处理等功能。

不管你是全文检索和 Elasticsearch 的初学者，还是使用过 Elasticsearch，你都能从本书中有所收获。

◆ 著　[波兰] Rafał Kuć　Marek Rogoziński
译　蔡建斌
责任编辑　李松峰
执行编辑　李　静　仇祝平
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：18.25
字数：431千字　2015年 3 月第 1 版
印数：1－3 000册　2015年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-8404号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

前　　言

欢迎阅读本书的第2版，这一版不仅针对Elasticsearch的最新版本更新了内容，还添加了一些在第1版中遗漏的重要内容。阅读这本书，你将踏上Elasticsearch服务器提供的全文检索的精彩旅程。本书首先对Elasticsearch进行一般性介绍，其中包括如何启动和运行Elasticsearch、Elasticsearch的基本概念，以及如何以最基本的方式索引和搜索数据。

本书也将讨论被称为Querydsl的查询语言，通过它可以创建复杂的查询并过滤返回的结果。除了这些，你还将看到如何使用切面技术（faceting）基于查询结果来计算汇总数据，以及如何使用新引进的聚合框架（分析引擎，可以为你的数据赋予意义）。我们将共同实现自动完成功能，并学习如何使用Elasticsearch的空间搜索能力（spatial capability）和预搜索（prospective search）。

最后，这本书将向你展示Elasticsearch的管理API，如分片安置控制和集群处理等功能。

本书主要内容

第1章　Elasticsearch集群入门，介绍什么是全文检索、Apache Lucene、文本分析、如何运行和配置Elasticsearch。最后，还会说明如何以最基本的方式索引和搜索数据。

第2章　索引，展示索引的工作原理，如何创建索引结构，可以使用什么样的数据类型，如何加速索引，什么是段（segment），合并（merging）是如何工作的，什么是路由（routing）。

第3章　搜索，介绍Elasticsearch的全文搜索功能。我们讨论如何查询，查询的工作原理，有哪些基本查询和复合查询。除此之外，本章还将展示如何过滤查询结果，如何高亮显示以及修改查询结果的排序。

第4章　扩展索引结构，讨论如何索引更复杂的数据结构。本章讨论如何索引树状数据类型和关系型数据，以及修改索引的结构。

第5章　更好的搜索，涵盖Apache Lucene的评分功能，以及使用Elasticsearch的脚本功能和语言分析器如何影响评分 。

第6章　超越全文检索，详细介绍聚合框架的功能、切面以及如何使用Elasticsearch实现拼写

检查和自动完成功能。此外，读者将学会如何索引二进制文件、处理地理空间数据，以及高效处理大数据集。

第7章　深入Elasticsearch集群，讨论节点发现机制，恢复和时光之门（Gateway）模块，高查询和高索引用例场景下的模板和集群。

第8章　集群管理，涵盖Elasticsearch备份功能、集群监控、再平衡和移动分片。除此之外，你还会学到如何使用热身功能和别名，安装插件，以及使用更新API来更新集群设置。

学习本书的准备工作

这本书所有的例子和功能都是用Elasticsearch服务器1.0.0版本写的，此外，你需要一个用来发送HTTP请求的命令工具，比如cURL，它在大多数操作系统上都可用。请注意，本书中的所有例子都使用cURL。如果你想使用另一种工具，请注意修改HTTP请求的格式，以便适合你所选择的工具。

此外，某些章节可能需要额外的软件，例如Elasticsearch插件，需要时我们会明确提及。

本书读者对象

如果你是一个全文检索和Elasticsearch的初学者，那么本书就是为你准备的。你将学到Elasticsearch的基础知识，以及如何使用一些高级功能。

如果你已经知道并使用了Elasticsearch，仍然会发现本书很有趣，因为它通过例子和描述，很好地概述了Elasticsearch的所有功能。

如果你知道Apache Solr搜索引擎，那么这本书也可以用来比较Apache Solr和Elasticsearch的某些功能。了解一些Elasticsearch的知识后，你可能会发现它更适合你。

排版规范

在这本书中，你会发现一些不同的文本样式用以区别不同种类的信息。下面是这些样式的一些例子和解释。

- 楷体

用于表示新术语。

- 等宽字体

表示程序中使用的变量名、关键字。

代码段格式如下所示：

```
{
  "status" : 200,
  "name" : "es_server",
  "version" : {
    "number" : "1.0.0",
    "build_hash" : "a46900e9c72c0a623d71b54016357d5f94c8ea32",
    "build_timestamp" : "2014-02-12T16:18:34Z",
    "build_snapshot" : false,
    "lucene_version" : "4.6"
  },
  "tagline" : "You Know, for Search"
}
```

当我们希望你注意代码块中的某些部分时，相关的行或者文字会被加粗：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
          "index" : "analyzed", "similarity" : "BM25" },
        "contents" : { "type" : "string", "store" : "no",
          "index" : "analyzed", "similarity" : "BM25" }
      }
    }
  }
}
```

命令行输入或输出如下所示：

```
curl -XGET http://localhost:9200/blog/article/1
```

这个图标表示警告或需要特别注意的内容。

这个图标表示提示或者技巧。

读者反馈

欢迎提出反馈，你对本书有任何想法，喜欢它什么，不喜欢它什么，请让我们知道。要写出真正对大家有帮助的图书，读者的反馈很重要。

一般的反馈，请发送电子邮件至feedback@packtpub.com，并在邮件主题中包含书名。如果你有某个主题的专业知识，并且有兴趣写成或帮助促成一本书，请参考我们的作者指南http://www.packtpub.com/authors。

客户支持

现在，你是一位令我们自豪的Packt图书的拥有者，我们会尽全力帮你充分利用你手中的书。

下载示例代码

你可以用你的账户从http://www.packtpub.com下载所有已购买Packt图书的示例代码文件。如果你从其他地方购买本书，可以访问http://www.packtpub.com/support并注册，我们将通过电子邮件把文件发送给你。

勘误表

虽然我们已尽力确保本书内容正确，但出错仍旧在所难免。如果你在我们的书中发现错误，不管是文本还是代码，希望能告知我们，我们不胜感激。这样做，你可以使其他读者免受挫败，帮助我们改进本书的后续版本。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata提交，选择你的书，点击勘误表提交表单的链接，并输入详细说明。勘误一经核实，你的提交将被接受，此勘误将上传到本公司网站或添加到现有勘误表。从http://www.packtpub.com/support选择书名就可以查看现有的勘误表。

侵权行为

版权材料在互联网上的盗版是所有媒体都要面对的问题。Packt非常重视保护版权和许可证。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供位置地址或网站名称，以便我们可以寻求补救。

请把可疑盗版材料的链接发到copyright@packtpub.com。

非常感谢你帮助我们保护作者，以及保护我们给你带来有价值内容的能力。

问题

如果你对本书内容存有疑问，不管是哪个方面，都可以通过questions@packtpub.com联系我们，我们将尽最大努力来解决。

Rafał Kuć个人致谢

你手中的这本书是2013年年初的*Elasticsearch Server*一书的升级版。自那以后，Elasticsearch有了很大的变化，涉及集群处理和搜索方面的内容时，功能有了很多改进并增加了大量内容。完成*Mastering ElasticSearch*（涵盖这个优秀搜索服务器的0.90版）后，我们觉得用1.0版来更新我们的第一本书是个绝好的时机。同第1版一样，这一版也无法详细介绍所有主题。为了不让本书成为1000页的大部头，我们明确了哪些需要详细描述，哪些需要提及，哪些需要省略。不管怎样，我希望通过阅读本书，你能很容易地了解Elasticsearch和底层的Apache Lucene，并且能更方便快捷地得到想要的知识。

我要感谢家人在我坐在屏幕前无法陪伴他们的那段日子里，对我的支持和耐心。

我还要感谢Sematext的所有同事，尤其是Otis，他拿出了宝贵的时间，让我相信Sematext是适合我的公司。

最后，我要感谢所有参与创建、开发和维护Elasticsearch和Lucene项目的人，感谢他们的工作和热情。没有他们，就不会有这本书，开源的搜索技术也不会这么强大。

再次感谢大家！

Marek Rogoziński个人致谢

这是我们关于Elasticsearch的第三本书，也是第一本书的第2版，第1版出版于2013年。时隔不久，但这也是Elasticsearch改变的一年。不到一年前，我们使用0.20版，而现在，1.0.1版已经发布。这不仅是数字上的改变，Elasticsearch现在已经成为一个广为人知并广泛使用的软件，有内置的商业支持和生态系统，比如Logstash、Kibana或其他插件。该搜索服务器的功能也在不断增加。有些新的功能，比如聚合框架，开辟了新的用例。这是Elasticsearch的闪光点。这种发展导致上一版迅速过时，而跟上这些变化也是个很大的挑战。Beta发布候选和最终版本之间的差异，也导致我们在写作过程中多次修改。

现在，是时候说声谢谢了。

感谢所有参与创建Elasticsearch、Lucene以及所有围绕这些项目发布或者使用库和模块的人。

我还要感谢这本书的出版团队。首先，感谢那些纠正本书技术错误、拼写错误和含糊之处的人。非常感谢给我们发送评价或写建设性评论的人。我很惊讶也很受鼓舞的是，有人发现我们的工作是有用的。

最后但同样重要的是，感谢所有容忍我并且理解我经常没有时间的朋友。

www.PacktPub.com

支持文件、电子书、折扣优惠等

你可以访问www.PacktPub.com来获得图书的支持文件和相关下载。

你知道吗？Packt为每本已出版图书提供PDF和ePub格式的电子版。你可以到www.PacktPub.com更新电子书的版本，并且如果你购买了纸质书，那么可以享受电子书的折扣。更多细节，可以通过service@packtpub.com与我们联系。

在www.PacktPub.com，你还可以阅读很多免费的技术文章，注册即可得到一系列免费新闻，并可以收到Packt图书和电子书的独家折扣和优惠信息。

http://PacktLib.PacktPub.com

你需要为你的IT问题得到即时的解决方案吗？PacktLib是Packt的在线数字图书馆。这里，你可以访问、阅读和搜索Packt的所有图书。

为什么订阅

- 搜索Packt出版的每一本书；
- 对内容进行复制粘贴、打印和打标签；
- 按需从浏览器访问。

Packt账户的免费访问

如果你已经有www.PacktPub.com的Packt账户，可以使用它来访问PacktLib并阅读9本免费图书。只需要使用你的登录凭据直接访问。

目　　录

第 1 章 Elasticsearch集群入门

欢迎来到Elasticsearch的奇妙世界，它是优秀的全文检索和分析引擎。不管你对Elasticsearch和全文检索有没有经验，都不要紧。我们希望你可以通过这本书，学习并扩展Elasticsearch的知识。由于这本书也是为初学者准备的，我们决定先简单介绍一般性的全文检索概念，接着再简要概述Elasticsearch。

我们要做的第一件事就是安装Elasticsearch。与许多应用相同，你从安装和配置着手，并经常忘记这些步骤的重要性。我们会尽量引导你完成这些步骤，从而使你更容易记住要点。此外，我们将告诉你如何用最简单的方法来索引和检索数据，而不用陷入太多细节。读完本章，你将学到以下内容：

- 全文检索；
- 了解Apache Lucene；
- 文本分析；
- 学习Elasticsearch的基本概念；
- 安装和配置Elasticsearch；
- 使用Elasticsearch REST API来操纵数据；
- 使用基本的URI请求来搜索。

1.1 全文检索

在全文检索只为一小部分工程师所知的时代，我们大多数人使用SQL数据库来执行搜索操作。当然它至少在一定程度上没什么问题。然而，当你越钻越深，就会看到这种方法的局限，如缺乏扩展性、不够灵活、缺乏语言分析（当然SQL数据库的全文检索对此有所作为）等。于是Apache Lucene（http://lucene.apache.org）出现了，它的目标是提供一个全文检索的功能库。它非常快速，可扩展，并提供不同语言的分析能力。

1.1.1 Lucene词汇表和架构

深入介绍分析处理的细节之前，我们先介绍一下Apache Lucene的词汇表和整体架构，下面

是这个库的基本概念。

- 文档（document）：索引和搜索时使用的主要数据载体，包含一个或多个存有数据的字段。
- 字段（field）：文档的一部分，包含名称和值两部分。
- 词（term）：一个搜索单元，表示文本中的一个词。
- 标记（token）：表示在字段文本中出现的词，由这个词的文本、开始和结束偏移量以及类型组成。

Apache Lucene将所有信息写到一个称为倒排索引（inverted index）的结构中。不同于关系型数据库中表的处理方式，倒排索引建立索引中词和文档之间的映射。你可以把倒排索引看成这样一种数据结构，其中的数据是面向词而不是面向文档的。来看一个简单的例子。我们有一些文档，只有它们的标题字段需要被索引，它们看起来如下所示：

- Elasticsearch Server 1.0 (document 1)；
- Mastering Elasticsearch (document 2)；
- Apache Solr 4 Cookbook (document 3)。

那么，简化版的索引可以看成是这样的：

词	计　数	文　档
1.0	1	<1>
4	1	<3>
Apache	1	<3>
Cookbook	1	<3>
Elasticsearch	2	<1>,<2>
Mastering	1	<2>
Server	1	<1>
Solr	1	<3>

每一个词指向包含它的文档编号。这样就可以执行一种非常高效且快速的搜索，比如基于词的查询。此外，每个词有一个计数，告诉Lucene该词出现的频率。

当然，Lucene实际创建的索引要比这个复杂得多，也先进得多，它创建的额外文件包含了词向量（term vector）、文档值（doc value）等信息。然而，到现在为止，你需要知道的是数据怎么组织，而不是具体怎么存储。

每个索引分为多个“写一次，读多次”（write once and read many time）的段（segment）。建立索引时，一个段写入磁盘后就不能再更新。因此，被删除文档的信息存储在一个单独的文件中，但该段自身不被更新。

然而，多个段可以通过段合并（segments merge）合并在一起。当强制段合并或者Lucene决定合并时，这些小段就会由Lucene合并成更大的一些段。合并需要I/O。然而一些信息需要清除，

因为在合并时，不再需要的信息将被删除（例如，被删除的文档）。除此之外，检索大段比检索存有相同数据的多个小段速度更快。这是因为在一般情况下，搜索只需将查询词与那些被编入索引的词相匹配。通过多个小段寻找和合并结果，显然会比让一个大段直接提供结果慢得多。

1.1.2 输入数据分析

当然，问题是，传入文档中的数据怎样转化成倒排索引，查询文本怎样变成可被搜索的词？这个数据转化的过程被称为分析。你可能希望某些字段经语言分析器处理，使得car和cars在索引中被视为同一个。另外，你可能希望另一些字段只用空格或者小写划分。

分析的工作由分析器完成，它由一个分词器（tokenizer）和零个或多个标记过滤器（token filter）组成，也可以有零个或多个字符映射器（character mapper）。

Lucene中的分词器把文本分割成多个标记，基本就是词加上一些额外信息，比如该词在原始文本中的位置和长度。分词器的处理结果称为标记流（token stream），它是一个接一个的标记，准备被过滤器处理。

除了分词器，Lucene分析器包含零个或多个标记过滤器，用来处理标记流中的标记。下面是一些过滤器的例子。

- 小写过滤器（lowercase filter）：把所有的标记变成小写。
- 同义词过滤器（synonyms filter）：基于基本的同义词规则，把一个标记换成另一个同义的标记。
- 多语言词干提取过滤器（multiple language stemming filter）：减少标记（实际上是标记中的文本部分），得到词根或者基本形式，即词干。

过滤器是一个接一个处理的。所以我们通过使用多个过滤器，几乎可以达到无限的分析可能性。

最后，字符映射器对未经分析的文本起作用，它们在分词器之前工作。因此，我们可以很容易地从文本的整体部分去除HTML标签而无需担心它们被标记。

索引和查询

我们可能想知道当使用Lucene和所有建立在它之上的软件时，上述所有功能对索引和查询的影响。建立索引时，Lucene会使用你选择的分析器来处理你的文档内容。当然，不同的字段可以使用不同的分析器，所以文档的名称字段可以和汇总字段做不同的分析。如果我们愿意，也可以不分析字段。

查询时，查询将被分析。但是，你也可以选择不分析。记住这一点很关键，因为一些Elasticsearch查询被分析，一些则不然。例如，前缀和词查询不被分析，匹配查询则被分析。可以在被分析查询和不被分析查询两者中选择非常有用。有时，你可能希望查询一个未经分析

的字段，而有时你则希望有全文搜索的分析。如果我们查询LightRed这个词，标准分析器分析这个查询后，会去查询light和red；如果我们使用不经分析的查询类型，则会明确地查询LightRed这个词。

关于索引和查询分析，你应该记住的是，索引应该和查询词匹配。如果它们不匹配，Lucene不会返回所需文档。比如，你在建立索引时使用了词干提取和小写，那你应该保证查询中的词也必须是词干和小写，否则你的查询不会返回任何结果。重要的是在索引和查询分析时，对所用标记过滤器保持相同的顺序，这样被分析出来的词是一样的。

1.1.3　评分和查询相关性

另外还有件现在还没提到的事，就是评分（scoring）。什么是文档的得分？得分是根据文档和查询的匹配度用计分公式计算的结果。默认情况下，Apache Lucene使用TF/IDF（term frequency/inverse document frequency，词频/逆向文档频率）评分机制，这是一种计算文档在我们查询上下文中相关度的算法。当然，它不是唯一可用的算法，2.2节将介绍其他算法。

> 如果你想阅读更多关于Apache Lucene TF/IDF评分公式的内容，请访问Apache Lucene Javadocs中的`TFIDFSimilarity`类，网址：http://lucene.apache.org/core/4_6_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html。

请记住，Elasticsearch和Lucene计算的分数值越高，意味着文档越相关。一些参数（比如boost）、不同的查询类型（3.3节将讨论这些查询类型）、不同的评分算法，都会影响得分的计算。

> 如果您想更深入地了解Apache Lucene评分是如何工作的、默认算法是什么、分数是如何计算的，请参阅我们的书*Mastering ElasticSearch*，Packt出版。

1.2　Elasticsearch 基础

Elasticsearch是由Shay Banon发起的一个开源搜索服务器项目，2010年2月发布。迄今，该项目已发展成为搜索和数据分析解决方案领域的主要一员，广泛应用于声名卓著或鲜为人知的搜索应用程序。此外，由于其分布式性质和实时功能，许多人把它作为文档数据库。

1.2.1　数据架构的主要概念

让我们过一遍Elasticsearch的基本概念。如果你已经熟悉Elasticsearch架构，可以跳过本节。如果你不熟悉这种架构，请考虑阅读本节。我们将提到本书其余部分会用到的关键字。

1. 索引

索引（index）是Elasticsearch对逻辑数据的逻辑存储，所以它可以分为更小的部分。你可以把索引看成关系型数据库的表。然而，索引的结构是为快速有效的全文索引准备的，特别是它不存储原始值。如果你知道MongoDB，可以把Elasticsearch的索引看成MongoDB里的一个集合。如果你熟悉CouchDB，可以把索引看成CouchDB数据库索引。Elasticsearch可以把索引存放在一台机器或者分散在多台服务器上，每个索引有一或多个分片（shard），每个分片可以有多个副本（replica）。

2. 文档

存储在Elasticsearch中的主要实体叫文档（document）。用关系型数据库来类比的话，一个文档相当于数据库表中的一行记录。当比较Elasticsearch中的文档和MongoDB中的文档，你会发现两者都可以有不同的结构，但Elasticsearch的文档中，相同字段必须有相同类型。这意味着，所有包含`title`字段的文档，`title`字段类型都必须一样，比如`string`。

文档由多个字段组成，每个字段可能多次出现在一个文档里，这样的字段叫多值字段（multivalued）。每个字段有类型，如文本、数值、日期等。字段类型也可以是复杂类型，一个字段包含其他子文档或者数组。字段类型在Elasticsearch中很重要，因为它给出了各种操作（如分析或排序）如何被执行的信息。幸好，这可以自动确定，然而，我们仍然建议使用映射。与关系型数据库不同，文档不需要有固定的结构，每个文档可以有不同的字段，此外，在程序开发期间，不必确定有哪些字段。当然，可以用模式强行规定文档结构。从客户端的角度看，文档是一个JSON对象（关于JSON格式的更多内容，参见http://en.wikipedia.org/wiki/JSON）。每个文档存储在一个索引中并有一个Elasticsearch自动生成的唯一标识符和文档类型。文档需要有对应文档类型的唯一标识符，这意味着在一个索引中，两个不同类型的文档可以有相同的唯一标识符。

3. 文档类型

在Elasticsearch中，一个索引对象可以存储很多不同用途的对象。例如，一个博客应用程序可以保存文章和评论。文档类型让我们轻易地区分单个索引中的不同对象。每个文档可以有不同的结构，但在实际部署中，将文件按类型区分对数据操作有很大帮助。当然，需要记住一个限制，不同的文档类型不能为相同的属性设置不同的类型。例如，在同一索引中的所有文档类型中，一个叫`title`的字段必须具有相同的类型。

4. 映射

在有关全文搜索基础知识部分，我们提到了分析的过程：为建索引和搜索准备输入文本。文档中的每个字段都必须根据不同类型做相应的分析。举例来说，对数值字段和从网页抓取的文本字段有不同的分析，比如前者的数字不应该按字母顺序排序，后者的第一步是忽略HTML标签，因为它们是无用的信息噪音。Elasticsearch在映射中存储有关字段的信息。每一个文档类型都有自己的映射，即使我们没有明确定义。

1.2.2　Elasticsearch主要概念

现在，我们已经知道Elasticsearch把数据存储在一个或多个索引上，每个索引包含各种类型的文档。我们也知道了每个文档有很多字段，映射定义了Elasticsearch如何对待这些字段。但还有更多，从一开始，Elasticsearch就被设计为能处理数以亿计的文档和每秒数以百计的搜索请求的分布式解决方案。这归功于几个重要的概念，我们现在将更详细地描述。

1. 节点和集群

Elasticsearch可以作为一个独立的单个搜索服务器。不过，为了能够处理大型数据集，实现容错和高可用性，Elasticsearch可以运行在许多互相合作的服务器上。这些服务器称为*集群*（cluster），形成集群的每个服务器称为*节点*（node）。

2. 分片

当有大量的文档时，由于内存的限制、硬盘能力、处理能力不足、无法足够快地响应客户端请求等，一个节点可能不够。在这种情况下，数据可以分为较小的称为*分片*（shard）的部分（其中每个分片都是一个独立的Apache Lucene索引）。每个分片可以放在不同的服务器上，因此，数据可以在集群的节点中传播。当你查询的索引分布在多个分片上时，Elasticsearch会把查询发送给每个相关的分片，并将结果合并在一起，而应用程序并不知道分片的存在。此外，多个分片可以加快索引。

3. 副本

为了提高查询吞吐量或实现高可用性，可以使用分片副本。*副本*（replica）只是一个分片的精确复制，每个分片可以有零个或多个副本。换句话说，Elasticsearch可以有许多相同的分片，其中之一被自动选择去更改索引操作。这种特殊的分片称为*主分片*（primary shard），其余称为*副本分片*（replica shard）。在主分片丢失时，例如该分片数据所在服务器不可用，集群将副本提升为新的主分片。

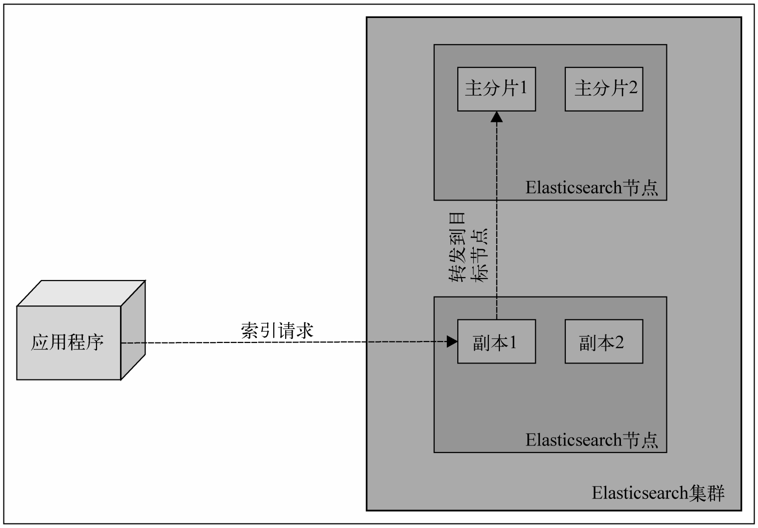
4. 时光之门

Elasticsearch处理许多节点。集群的状态由时光之门控制。默认情况下，每个节点都在本地存储这些信息，并且在节点中同步。我们将在7.2节详细讨论时光之门模块。

1.2.3　索引建立和搜索

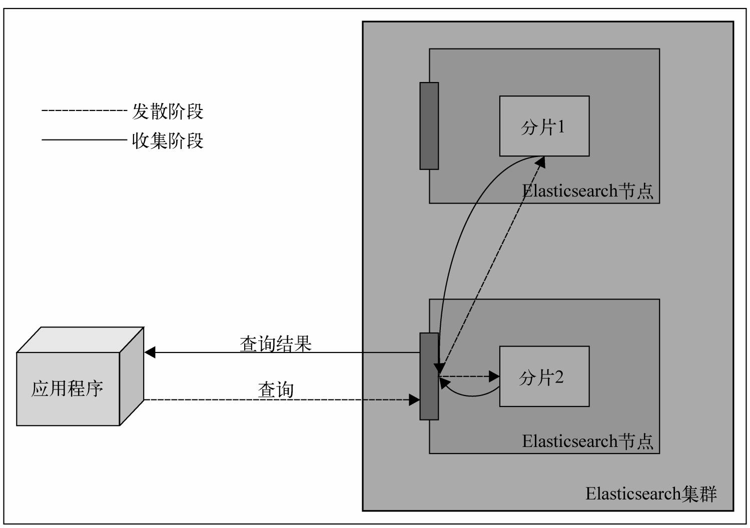
你可能会问实际上如何把所有的索引、分片和副本绑在单个环境里。理论上，当你必须知道你的文档在哪，哪台服务器、哪个分片上时，从集群获取数据非常困难。更为困难的是当一个搜索需要返回的文档分布在集群中不同节点的不同分片上时。这确实是一个复杂的问题，好在我们

不需要关心，它由Elasticsearch本身自动处理。来看看下图：



发送一个新的文档给集群时，你指定一个目标索引并发送给它的任意一个节点。这个节点知道目标索引有多少分片，并且能够确定哪个分片应该用来存储你的文档。可以更改Elasticsearch的这个行为。2.6.3节将对此进行讨论。现在你需要记住的重要信息是，Elasticsearch使用文档的唯一标识符来计算文档应该被放到哪个分片中。索引请求发送到一个节点后，该节点会转发文档到持有相关分片的目标节点中。

现在来看看关于执行搜索请求的图：



尝试用文档标识符来获取文档时，发送查询到一个节点，该节点使用同样的路由算法来决定持有文档的节点和分片，然后转发查询，获取结果，并把结果发送给你。另一方面，查询过程更为复杂。除非使用了路由，查询将直接转发到单个分片，否则，收到查询请求的节点会把查询转发给保存了属于给定索引的分片的所有节点，并要求与查询匹配的文档的最少信息（默认情况下是标识符和得分）。这个过程称为发散阶段（scatter phase）。收到这些信息后，该聚合节点（收到客户端请求的节点）对结果排序，并发送第2个请求来获取结果列表所需的文档（除了标识符和得分以外的所有信息）。这个阶段称为收集阶段（gather phase）。这个阶段执行完毕后，结果返回到客户端。

现在问题来了，在前面描述的过程中，副本扮演了什么角色呢？在建立索引时，副本只作为额外的位置来存储数据。当执行查询时，默认情况下，Elasticsearch会尽量平衡分片和它的副本之间的负载，使它们承受均衡的压力。此外，记住我们可以改变该行为。3.2节将对此进行讨论。

1.3　安装并配置集群

有几个安装Elasticsearch所需的步骤，接下来几节将详细说明。

1.3.1　安装Java

为了建立Elasticsearch，第一步是确保正确安装Java SE环境。Elasticsearch需要Java 6或更高版本。你可以从http://www.oracle.com/technetwork/java/javase/downloads/index.html下载。如果你想，也可以使用OpenJDK（http://openjdk.java.net/）。当然你可以使用Java 6，但它已经没有补丁的支持，所以建议安装Java 7。

1.3.2　安装Elasticsearch

从http://www.elasticsearch.org/download/下载，解压。选择最新的稳定版本，安装完毕。

> 写这本书时，我们用的是Elasticsearch 1.0.0 GA。这意味着，我们已经跳过一些被标记为过时（deprecated）的属性的描述，它们已经或将在未来的Elasticsearch版本中被移除。

与Elasticsearch交互的主要接口是基于HTTP协议和REST的。这意味着你甚至可以使用Web浏览器来完成基本的查询和请求，但对于更复杂的情况，你需要额外的命令行工具，比如cURL。如果你使用Linux或OS X命令，cURL已经可用了。如果你使用Windows，可以从http://curl.haxx.se/download.html下载。

1.3.3　在Linux上用二进制包安装Elasticsearch

安装Elasticsearch的另一个方法是使用提供的二进制包，RPM或DEB，视你的Linux发行版而定。这些二进制包可以在http://www.elasticsearch.org/download/找到。

1. 使用RPM包安装Elasticsearch

下载RPM包后，你只需执行如下命令：

```
sudo yum elasticsearch-1.0.0.noarch.rpm
```

就这么简单。如果一切顺利，Elasticsearch应该安装好了，配置文件应该在/etc/sysconfig/elasticsearch中。如你的操作系统基于红帽，应该可以使用/etc/init.d/elasticsearch下的`init`脚本。如果你的操作系统是SUSE Linux，可以使用/bin下的systemctl文件来启动和停止Elasticsearch服务。

2. 使用DEB包安装Elasticsearch

下载DEB包后，只需执行如下命令：

```
sudo dpkg -i elasticsearch-1.0.0.deb
```

就这么简单。如果一切顺利，Elasticsearch应该安装成功，配置文件存在/etc/elasticsearch/elasticsearch.yml。/etc/init.d/elasticsearch下的`init`脚本可以用来启动和停止Elasticsearch。此外，/etc/default/elasticsearch下的文件包含了环境设置。

1.3.4　目录布局

现在，到新创建的目录中。应该可以看到下面的目录结构：

目　　录	描　　述
bin	运行Elasticsearch实例和插件管理所需的脚本
config	配置文件所在的目录
lib	Elasticsearch使用的库

Elasticsearch启动后，会创建如下目录（如果目录不存在）：

目　　录	描　　述
data	Elasticsearch使用的所有数据的存储位置
logs	关于事件和错误记录的文件
plugins	存储所安装插件的地方
work	Elasticsearch使用的临时文件

1.3.5　配置Elasticsearch

Elasticsearch很容易入门，这是它越来越流行的原因之一，当然，不是全部原因。因为已为简单的环境配置了合理的默认值和自动设置，我们可以跳过配置，不需改变我们的配置文件中的任意一行而直接走到下一章。然而，为了真正理解Elasticsearch，学习一些可用的设置还是值得的。

现在，来探讨Elasticsearch的tar.gz存档提供的默认目录和文件的布局。整个配置位于config目录下，可以看到两个文件：elasticsearch.yml（或elasticsearch.json，如果有的话会被使用）和logging.yml。第一个文件负责设置服务器的默认配置值。重要的是，因为一些配置值可以在运行时更改，也可作为集群状态的一部分被保留，所以这个文件中的值可能不准确。有两个值不能在运行时更改，分别是`cluster.name`和`node.name`。

`cluster.name`属性保存集群的名字，不同的集群用名字来区分，配置成相同集群名字的各个节点形成一个集群。

`node.name`是实例（该节点）的名字，可以不定义此参数，这时，Elasticsearch自动选择一个唯一的名称。注意，此名称是每次启动时选择的，所以在每次重启后名称可能都不一样。在很长的时间区间或者重启过后，需要在API中提及具体实例名称，或者用监控工具查看节点，自定义一个名称还是很有帮助的。给你的节点想一个描述性的名字吧。

文件中的其他参数有很好的注释，所以建议你看看。不要担心不理解那些解释。希望在读完下面几章后，一切都变得清晰起来。

> 记住，大多数在elasticsearch.yml文件中设置的参数都可以用Elasticsearch REST API来覆盖。8.8节将介绍这些API。

第2个文件（logging.yml）定义了多少信息写入系统日志，定义了日志文件，并定期创建新文件。只有在调整监控、备份方案或系统调试时，才需要修改。然而如果想有一份更详细的日志，就需要相应调整。

我们保留这些配置文件不动。配置的一个重要部分是调整你的操作系统。在建立索引时，尤其是有很多分片和副本的情况下，Elasticsearch将创建很多文件。所以，系统不能限制打开的文件描述符小于32 000个。在Linux上，一般在/etc/security/limits.conf中修改，当前的值可以用`ulimit`命令来查看。如果达到极限，Elasticsearch将无法创建新的文件，所以合并会失败，索引会失败，新的索引无法创建。

下一组设定关联到单个Elasticsearch实例的Java虚拟机（JVM）的堆内存限制。对小型部署来说，默认的内存限制（1024 M）就足够了，但对于大型项目不够。如果你在日志文件中发现

OutOfMemoryError异常的条目，把ES_HEAP_SIZE变量设置到大于1024。当选择分配给JVM的合适内存大小时，记住，通常不应该分配超过50%的系统总内存。不过，所有的规则都有例外，稍后将更详细地讨论，但你应该经常监控JVM堆的使用量，需要时调整。

1.3.6　运行Elasticsearch

运行刚刚下载并解压的ZIP包，转到bin目录，然后根据不同的操作系统，运行如下命令。

- Linux或OS X：./elasticsearch。
- Windows：elasticsearch.bat。

恭喜你！现在把Elasticsearch启动并运行起来了。它在工作时一般使用2个端口号：第1个是使用HTTP协议与REST API通信的端口，第2个是传输模块（transport module），是用来在集群内以及Java客户端和集群之间通信的端口。HTTP API的默认端口号是9200，所以可以在浏览器中打开http://127.0.0.1:9200来检查搜索是否就绪，浏览器将显示类似下面这样的代码片段：

```
{
  "status" : 200,
  "name" : "es_server",
  "version" : {
    "number" : "1.0.0",
    "build_hash" : "a46900e9c72c0a623d71b54016357d5f94c8ea32",
    "build_timestamp" : "2014-02-12T16:18:34Z",
    "build_snapshot" : false,
    "lucene_version" : "4.6"
  },
  "tagline" : "You Know, for Search"
}
```

输出是JSON结构的。如果你还不熟悉JSON（JavaScript Object Notation），请花几分钟阅读http://en.wikipedia.org/wiki/JSON上的这篇文章。

> Elasticsearch很聪明。如果默认端口不可用，引擎将绑定到下一个可用端口，你可以在启动时的控制台上找到如下相关信息：
>
> ```
> [2013-11-16 11:56:12,101][INFO ][http] [Red Lotus]
>  bound_address {inet[/0:0:0:0:0:0:0:0%0:9200]},
>    publish_address {inet[/192.168.1.101:9200]}
> ```
>
> 注意[http]的片段。Elasticsearch使用了一些端口完成各种任务。我们所使用的接口是由HTTP模块处理的。

现在，将使用cURL程序。例如，要检查集群健康度，会使用以下命令：

```
curl -XGET http://127.0.0.1:9200/_cluster/health?pretty
```

参数-X是一个请求方法，默认值是GET（所以在上面的例子中，可以忽略此参数）。暂时不要担心GET这个值，本章的后面将更详细地描述它。

作为一个标准，API返回的JSON对象信息里，换行符是被省略的，在请求中加上pretty参数是强制Elasticsearch在响应中加上换行符，使之更可读。你可以试着去掉pretty参数运行上面的请求，看看有什么不同。

Elasticsearch在中小型应用程序中非常有用，但它的初衷是建成大型集群。所以，现在来建立由两个节点组成的大的集群。解压Elasticsearch到另一个目录，然后运行第二个实例。我们会在日志中看到如下内容：

```
[2013-11-16 11:55:16,767][INFO ][cluster.service]
[Stane, Obadiah] detected_master [Martha Johansson]
[vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]], added {[Martha Johansson]
[vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]],}, reason: zen-disco-receive(from master
[[Martha Johansson][vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]]])
```

这意味着我们的第二个实例（名字为Stane,Obadiah）检测到了之前运行的实例（名字为Martha Johansson）。这里，Elasticsearch自动形成了一个新的双节点集群。

> 请注意，在某些系统上，防火墙软件默认自动打开，可能导致节点无法找到其他节点。

1.3.7　关掉Elasticsearch

尽管我们期望集群或节点完美地一直运行下去，但仍可能需要正确地重启或者关闭，比如，为了维护。下面是三种可以关闭Elasticsearch的方法。

- 如果节点是连接到控制台，按下Ctrl+C。
- 第二种选择是通过发送TERM信号杀掉服务器进程（参考Linux上的kill命令和Windows上的任务管理器）。
- 第三种方法是使用REST API。

现在着重介绍第三种方法。可以执行以下命令来关掉整个集群：

```
curl -XPOST http://localhost:9200/_cluster/nodes/_shutdown
```

为关闭单一节点，假如节点标识符是BlrmMvBdSKiCeYGsiHijdg，可以执行下面的命令：

```
curl -XPOST
http://localhost:9200/_cluster/nodes/BlrmMvBdSKiCeYGsiHijdg/_shutdown
```

节点的标识符可以在日志中看到，或者使用_cluster/nodes API，命令如下：

```
curl -XGET http://localhost:9200/_cluster/nodes/
```

1.3.8　Elasticsearch作为系统服务运行

Elasticsearch 1.0可以作为服务运行在基于Linux的系统和基于Windows的系统上。

1. 在Linux上运行系统服务

如果是从提供的二进制包安装的Elasticsearch，你已经完成了，什么都不用担心。但是，如果你刚刚下载归档文件，解压到所选择的目录，就需要做一些额外的工作。为了将Elasticsearch安装成一个Linux系统服务，将使用Elasticsearch service wrapper，你可以从https://github.com/elasticsearch/elasticsearch-servicewrapper下载。

来看看使用Elasticsearch service wrapper建立Elasticsearch Linux服务的步骤。首先，执行以下命令来下载这个wrapper：

```
curl -L http://github.com/elasticsearch/elasticsearch-
  servicewrapper/tarball/master  | tar -xz
```

假设Elasticsearch已经安装在/usr/local/share/elasticsearch下，执行如下命令来移动所需的wrapper文件：

```
sudo mv *servicewrapper*/service/usr/local/share/elasticsearch/bin/
```

执行如下命令来移除剩余的文件

```
rm -Rf *servicewrapper*
```

最后，通过执行install命令来安装服务：

```
sudo /usr/local/share/elasticsearch/bin/service/elasticsearch install
```

在这之后，需要创建一个符号链接指向/usr/local/bin/rcelasticsearch下的/usr/local/share/elasticsearch/bin/service/elasticsearch脚本。可通过运行如下命令来实现：

```
sudo ln -s 'readlink  -f
  /usr/local/share/elasticsearch/bin/service/elasticsearch'
    /usr/local/bin/rcelasticsearch
```

就这样。如果你想启动Elasticsearch，执行如下命令：

```
/etc/init.d/elasticsearch start
```

2. 在Windows上运行系统服务

在Windows下把Elasticsearch安装为系统服务非常容易，你只需转到Elasticsearch的安装目录，

到bin子目录下，执行:

```
service.bat install
```

你会被问及操作权限，允许脚本运行，Elasticsearch就被安装成一个Windows服务。

如果你想看看所有被service.bat脚本文件暴露出来的命令，在相同目录下执行：

```
service.bat
```

例如，为了启动Elasticsearch，可执行如下命令：

```
service.bat start
```

1.4　用 REST API 操作数据

Elasticsearch REST API可用于各种任务。有了它，可以管理索引，更改实例参数，检查节点和群集状态，索引数据，搜索数据或者通过GET API检索文档。但是现在，我们将集中在API中的CRUD（create-retrieve-update-delete，增删改查）部分，它让我们能像使用NoSQL数据库一样使用Elasticsearch。

1.4.1　理解Elasticsearch的RESTful API

在一个类REST的架构中，每个请求都指向地址路径所表示的一个具体对象。如果/books/是一个图书馆中图书列表的引用，/books/1则引用ID为1的那本书。注意这些对象可以嵌套，/books/1/chapter/6表示图书馆的第一本书的第6章，等等。我们的API调用有个主题。我们想执行的操作（比如GET或POST操作）怎么样？请求类型就是用来指定这个的。HTTP协议给出了可以在API调用中用作动词的一组相当长的类型。合乎逻辑的选择是，GET用来获得请求对象的当前状态，POST来改变对象的当前状态，PUT创建一个对象，而DELETE销毁对象，另外还有个HEAD请求仅仅用来获取对象的基础信息。

现在来看看在1.3.7节中讨论的如下操作例子，一切都应该更容易理解。

- GET http://localhost:9000/：这个命令用来获取Elasticsearch的基本信息。
- GET http://localhost:9200/_cluster/state/nodes/：这个命令获取集群中节点的信息。
- POST http://localhost:9200/_cluster/nodes/_shutdown：这个命令向集群中所有节点发送一个shutdown请求。

我们现在至少知道了REST的一般概念。你可以在http://en.wikipedia.org/wiki/Representational_state_transfer上阅读更多关于REST的信息。现在，可以继续学习如何使用Elasticsearch API来存储、

读取、修改和删除数据。

1.4.2 在Elasticsearch中存储数据

我们已经讨论过，在Elasticsearch中，所有的数据，即每个文档，都有定义好的索引和类型。每个文档可以包含一个或多个字段来保存数据。首先展示如何使用Elasticsearch为一个简单文档建立索引。

1.4.3 新建文档

现在，尝试索引一些文档。例如，为博客建立某种内容管理系统（CMS）。文章（article）是博客中的一个实体。

使用JSON标记，一个文档可以如下所示的例子来表示：

```
{
  "id": "1",
  "title": "New version of Elasticsearch released!",
  "content": "Version 1.0 released today!",
  "priority": 10,
  "tags": ["announce", "elasticsearch", "release"]
}
```

可以看到，JSON文档包含一组字段，每个字段可以有不同的形式。在以上示例中，我们有数字（`priority`）、文本（`title`）和字符串数组（`tags`）。以下示例将展示其他类型。如本章前面所述，Elasticsearch能猜出这些类型（因为JSON是半类型化的，例如，数字没有放在引号中），并自动定制这些数据在其内部结构中如何存储。

当然，我们希望为示例文档建立索引，并使其可用于搜索。我们将使用一个名为`blog`的索引和名为`article`的类型。为了把示例文档以给定类型、标识符为1建立在索引中，执行以下命令：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New version of
Elasticsearch released!", content": "Version 1.0 released today!", "tags": ["announce",
"elasticsearch", "release"] }'
```

注意cURL命令的一个新选项：`-d`参数。此选项的值是将作为请求负载的文本，也即请求主体（request body）。这样，我们可以发送附加信息，如文档定义。同时，注意唯一标识符（1）是放在URL，而不是请求主体中。使用HTTP `PUT`请求时，如果省略此标识符，该请求将返回以下错误：

```
No handler found for uri [/blog/article/] and method [PUT]
```

如果一切正确，Elasticsearch会返回一个JSON响应，与如下输出类似：

```
{
  "_index":"blog",
  "_type":"article",
  "_id":"1",
  "_version":1
}
```

前面的响应包含了此次操作状态的信息，并显示一个新的文档放在哪里，还包含了有关文档的唯一标识符（_id）和当前版本（_version）的信息。版本将由Elasticsearch每次更新时自动递增。

标识符的自动创建

在上面的示例中，我们自己指定了文档标识符。然而，Elasticsearch可以自动生成它。这似乎很方便，但只有当该索引是唯一的数据来源时，才能这么做。如果使用一个数据库来存储数据，用Elasticsearch全文搜索，那数据同步将会被阻碍，除非在数据库中也存储生成的标识符。使用HTTP POST请求类型并且不在URL中指定标识符，就可以生成一个唯一标识符。例如，看下面的命令：

```
curl -XPOST http://localhost:9200/blog/article/ -d '{"title": "New version of
Elasticsearch released!", "content": "Version 1.0  released today!", "tags":
["announce", "elasticsearch", "release"] }'
```

注意，要使用HTTP POST方法，而不是前面示例中的PUT方法。参考前面关于REST动词的描述，我们想更改列表中的文档索引，而不是创建一个新的实体，所以使用POST而不是PUT。服务器应该返回类似下面的响应：

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "XQmdeSe_RVamFgRHMqcZQg",
  "_version" : 1
}
```

注意加粗的那一行，这是一个Elasticsearch自动生成的标识符。

1.4.4　检索文档

我们已经将实例存储在了文档中，现在尝试通过标识符检索。首先执行以下命令：

```
curl -XGET http://localhost:9200/blog/article/1
```

Elasticsearch将返回类似下面的响应：

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
```

```
  "_version" : 1,
  "exists" : true,
  "_source" : {
    "title": "New version of Elasticsearch released!",
    "content": "Version 1.0 released today!",
    "tags": ["announce", "elasticsearch", "release"]
  }
```

在前面的响应中，除了索引、类型、标识符和版本，还可以看到说明"发现文件存在"（exists属性）以及此文档来源（_source属性）的信息。如果没有找到文档，得到的响应如下所示：

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "9999",
  "exists" : false
}
```

因为没有找到文档，当然也就没有版本或来源的信息。

1.4.5　更新文档

更新索引中的文档是一项更复杂的任务。在内部，Elasticsearch必须首先获取文档，从_source属性获得数据，删除旧的文件，更改_source属性，然后把它作为新的文档来索引。它如此复杂，因为信息一旦在Lucene的倒排索引中存储，就不能再被更改。Elasticsearch通过一个带_update参数的脚本来实现它。这样就可以做比简单修改字段更加复杂的文档转换。下面用简单的例子看看的工作原理。

请记住之前建立的博客文章索引。为了更改其content字段，运行以下命令：

```
curl -XPOST http://localhost:9200/blog/article/1/_update -d '{
  "script": "ctx._source.content = \"new content\""
}'
```

Elasticsearch将返回如下响应：

```
{"_index":"blog","_type":"article","_id":"1","_version":2}
```

看上去更新操作执行成功了。为了确定，我们用它的标识符检索一下，执行如下命令：

```
curl -XGET http://localhost:9200/blog/article/1
```

Elasticsearch的响应应该包含修改过的content字段，事实的确如此，它包含如下信息：

```
{
"_index" : "blog",
"_type" : "article",
"_id" : "1",
```

```
"_version" : 2,
"exists" : true,
"_source" : {
  "title":"New version of Elasticsearch released!",
  "content":"new content",
  "tags":["announce","elasticsearch","release"]
}
```

Elasticsearch修改了文章的content和该文档的版本号。注意，不必发送整个文档，只需发送改变的部分。但是请记住，为了使用更新功能，需要使用_source字段，2.4节将描述如何使用_source字段。

关于文档更新，还有一点，如果你的脚本需要更新文档的一个字段，你可以设置一个值用来处理文档中没有该字段的情况。例如，想增加文档中的counter字段，而该字段不存在，你可以在请求中使用upsert节来提供字段的默认值。看下面的例子：

```
curl -XPOST http://localhost:9200/blog/article/1/_update -d '{
  "script": "ctx._source.counter += 1",
  "upsert": {
    "counter" : 0
  }
}'
```

执行这个示例，Elasticsearch会在示例文档中添加一个值为0的counter字段。这是因为我们的文档没有counter字段，而我们在更新请求中指定了upsert节。

1.4.6　删除文档

我们已经看到如何创建（PUT）、检索（GET）和更新文档，不难猜到，删除文档的过程是类似的：需要使用DELETE请求类型发送一个适当的HTTP请求。例如，要删除示例文档，运行以下命令：

```
curl -XDELETE http://localhost:9200/blog/article/1
```

Elasticsearch的响应如下所示：

```
{"found":true,"_index":"blog","_type":"article","_id":"1","_version":3}
```

这意味着我们找到并删除了该文档。

现在可以利用CRUD操作。我们已经可以使用Elasticsearch作为一个简单的键值存储来创建应用程序。但这仅仅是开始！

1.4.7　版本控制

在提供的例子中，你可能注意到了文档的版本信息，它看起来如下所示：

```
"_version" : 1
```

仔细观察，你会发现在更新相同标识符的文档后，这个版本是递增的。默认情况下，Elasticsearch在添加、更改或删除文档时都会递增版本号。除了告诉我们对文档所做更改的次数，还能够实现乐观锁(optimistic locking，http://en.wikipedia.org/wiki/Optimistic_concurrency_control)。这允许我们在并发处理同一文档时避免问题。例如，在两个不同的应用程序中读取相同的文档，分别修改它，然后尝试更新到Elasticsearch。没有版本控制，我们将看到最后更新的版本。使用乐观锁，Elasticsearch保证数据的准确性，尝试写入一个已更改的文档将会失败。

1. 版本控制的一个例子

我们来看一个使用版本控制的示例。假设要删除library索引中类型为book、id为1的文档。我们也要确保如果文档没有更新，则删除操作成功。需要做的是添加一个值为1的version参数，如下所示：

```
curl -XDELETE 'localhost:9200/library/book/1?version=1'
```

如果索引中文档的版本不等于1，Elasticsearch将返回如下错误：

```
{
  "error": "VersionConflictEngineException[[library][4] [book][1]:
    version conflict, current [2], provided [1]]",
   "status": 409
}
```

在我们的示例中，Elasticsearch比较我们声明的版本号和在Elasticsearch中文档的版本号，发现不一样，所以操作失败。

2. 使用外部系统提供的版本

Elasticsearch也可基于我们提供给它的版本号。在版本存储在外部系统时，这是必要的。这种情况下，当你新索引一个文档时，应该如上面的示例一样提供一个version参数。这时，Elasticsearch将只检查提供的版本是否比当前保存在索引中的版本大(大多少并不重要)，如果是，操作成功，否则将失败。为了告诉Elasticsearch我们要使用外部版本跟踪，除了version参数外，还需要添加version_type=external参数。

例如，在系统添加文档版本123456，将运行如下命令：

```
curl -XPUT 'localhost:9200/library/book/1?version=123456' -d {...}
```

> 即使文档被移除后，Elasticsearch仍然可以检查版本号。这是因为Elasticsearch保留了删除文档的版本信息。默认情况下，此信息在删除的60秒内可用。可以通过修改index.gc_deletes配置参数来更改这个值。

1.5　使用 URI 请求查询来搜索

进入Elasticsearch查询的详细信息之前，先使用其中简单的URI请求来搜索。当然，第3章将扩展使用Elasticsearch搜索的知识，但是现在，先使用最简单的方法。

1.5.1　示例数据

本节将创建一个简单的索引，它有两个文档类型。为此，运行以下命令：

```
curl -XPOST 'localhost:9200/books/es/1' -d '{"title":"Elasticsearch Server",
  "published": 2013}'
curl -XPOST 'localhost:9200/books/es/2' -d '{"title":"Mastering Elasticsearch",
  "published": 2013}'
curl -XPOST 'localhost:9200/books/solr/1' -d '{"title":"Apache Solr 4 Cookbook",
  "published": 2012}'
```

运行上述命令将创建books索引，该索引包含两种类型：es和solr。Title和published字段将被索引。如果你想检查，可以通过运行以下命令映射API，2.2节将讨论映射：

```
curl -XGET 'localhost:9200/books/_mapping?pretty'
```

Elasticsearch将返回整个索引的所有映射。

1.5.2　URI请求

Elasticsearch的所有查询都发送到_search端点。你可以搜索单个或多个索引，也可以将搜索范围缩小到给定的一个或多个文档类型。例如，为了寻找books索引，运行以下命令：

```
curl -XGET 'localhost:9200/books/_search?pretty'
```

如果还有另一个索引叫clients，也可对这两个索引执行一个查询：

```
curl -XGET 'localhost:9200/books,clients/_search?pretty'
```

以同样的方式，还可以选择搜索时要使用的类型。如果只想在books索引的es类型中搜索，将运行如下命令：

```
curl -XGET 'localhost:9200/books/es/_search?pretty'
```

> 请记住，为了搜索一个给定的类型，需要指定一个或多个索引。如果要寻找任意索引，只需要设置星号（*）为索引名称，或忽略索引名称。Elasticsearch在选择索引名称时支持相当丰富的语义。如果你有兴趣，请参考http://www.lasticsearch.org/guide/en/elasticsearch/reference/current/multi-index.html。

还可以省略索引和类型来搜索所有索引。例如，以下命令将搜索集群中的所有数据：

```
curl -XGET 'localhost:9200/_search?pretty'
```

1. Elasticsearch查询响应

假设想找到books索引中title字段包含elasticsearch一词的所有文档，可以运行以下查询：

```
curl -XGET
  'localhost:9200/books/_search?pretty&q=title:elasticsearch'
```

Elasticsearch返回的响应如下所示：

```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.625,
    "hits" : [ {
      "_index" : "books",
      "_type" : "es",
      "_id" : "1",
      "_score" : 0.625, "_source" : {"title":"Elasticsearch Server",
        "published": 2013}
    }, {
      "_index" : "books",
      "_type" : "es",
      "_id" : "2",
      "_score" : 0.19178301, "_source" : {"title":"Mastering
        Elasticsearch", "published": 2013}
    } ]
  }
}
```

响应的第一部分告诉我们该请求花了多少时间（took属性，单位是毫秒），有没有超时（timed_out属性），执行请求时查询的分片信息，包括查询的分片数量（_shards对象的total属性）、成功返回结果的分片数量（_shards对象的successful属性）、失败的分片数量（_shards对象的failed属性）。如果查询执行时间比预想的更长，它可能会超时（可以使用timeout参数指定查询的最大执行时间）。可以使用超时参数，指定最大查询执行时间。失败的分片意味着分片出了问题或在执行搜索时不可用。

当然，上述信息很有用，但是通常我们对hits对象中返回的结果感兴趣。我们有查询返回的文档总数（total属性）和计算所得的最高分（max_score属性），还有包含返回文档的hits

数组。在本例中，每个返回的文档包含索引（_index属性）、类型（_type属性）、标识符（_id属性）、得分（_score属性）和_source字段（通常，这是发送到索引的JSON对象。这一内容将在2.4节讨论）。

2. 查询分析

你可能觉得奇怪为什么上一节运行的查询可以返回结果。用Elasticsearch建立索引，然后用elasticsearch来执行查询，虽然大小写不同，还是可以找到相关文档，原因就是查询分析。在建立索引时，底层的Lucene库根据Elasticsearch配置文件分析文档并建立索引数据。默认情况下，Elasticsearch会告诉Lucene对基于字符串的数据和数字都做索引和分析。查询阶段也一样，因为URI请求查询会映射到query_string查询（将在第3章讨论），Elasticsearch会分析它。

使用索引分析API（indices analyze API，http://www.elasticsearch.org/guide/en/ elasticsearch/eference/current/indices-analyze.html），可以看到分析过程是怎样的，在建立索引时发生了什么，在查询阶段又发生了什么。

为了看到title字段上的短语“Elasticsearch Server”建立的索引具体是什么，可以执行以下命令：

```
curl -XGET 'localhost:9200/books/_analyze?field=title' -d
  'Elasticsearch Server'
```

响应如下：

```
{
  "tokens" : [ {
    "token" : "elasticsearch",
    "start_offset" : 0,
    "end_offset" : 13,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "server",
    "start_offset" : 14,
    "end_offset" : 20,
    "type" : "<ALPHANUM>",
    "position" : 2
  } ]
}
```

可以看到，Elasticsearch把文本划分为两个词，第一个标记值（token value）为elasti search，第二个标记值为server。

现在看看查询文本是如何被分析的，运行以下命令：

```
curl -XGET 'localhost:9200/books/_analyze?pretty&field=title' -d
  'elasticsearch'
```

响应如下：

```
{
  "tokens" : [ {
    "token" : "elasticsearch",
    "start_offset" : 0,
    "end_offset" : 13,
    "type" : "<ALPHANUM>",
    "position" : 1
  } ]
}
```

可以看到，这个词和传到查询的原始值是一样的。我们不会详细介绍Lucene查询以及查询解析器如何构建查询，但总地来说，分析之后的索引词和分析之后查询词是一样的，因此，该文档与查询匹配并作为结果返回。

3. URI查询中的字符参数

有几个参数，可以用来控制URI查询行为，现在来讨论一下。查询中的每个参数应加上&字符，如以下示例所示：

```
curl -XGET
  'localhost:9200/books/_search?pretty&q=published:
  2013&df=title&explain= rue&default_operator=AND'
```

请记得'字符，因为在类Linux系统上，&字符会被Linux shell解析。

(1) 查询

参数q用来指定我们希望文件匹配的查询条件。可以使用Lucene查询语法来指定查询，1.5.3节会描述。例如，一个简单的查询可能类似`q=title:elasticsearch`。

(2) 默认查询字段

使用`df`参数，可以指定在q参数中没有字段时应该默认使用的字段。默认情况下，将使用`_all`字段。Elasticsearch把其他所有字段的内容复制到`_all`字段。2.4节将更深入地讨论。一个`df`参数的例子是`df=title`。

(3) 分析器

可以将`analyzer`属性定义用于分析查询的分析器名称。默认情况下，索引阶段对字段内容做分析的分析器将用来分析我们的查询。

(4) 默认操作符

`Default_operator`属性可以设置成`OR`或`AND`，用来指定用于查询的默认布尔运算符。默认情况下，它设置为`OR`，意味着只要有一个查询条件匹配，就将返回文档。此参数设置为`AND`时，所有查询条件都匹配时才会返回文档。

(5) 查询解释

如果将explain参数设置为true，Elasticsearch将在结果的每个文档里包括额外的解释信息，如文档是从哪个分片上获取的、计算得分的详细信息（5.7节将深入讨论）。记住，不要在正常的搜索查询中设置explain为true，因为它需要额外的资源并使查询的性能下降。下面的代码是一个例子：

```
{
  "_shard" : 3,
  "_node" : "kyuzK62NQcGJyhc2gI1P2w",
  "_index" : "books",
  "_type" : "es",
  "_id" : "2",
  "_score" : 0.19178301, "_source" : {"title":"Mastering
    Elasticsearch", "published": 2013},
  "_explanation" : {
    "value" : 0.19178301,
    "description" : "weight(title:elasticsearch in 0)
      [PerFieldSimilarity], result of:",
    "details" : [ {
      "value" : 0.19178301,
      "description" : "fieldWeight in 0, product of:",
      "details" : [ {
        "value" : 1.0,
        "description" : "tf(freq=1.0), with freq of:",
        "details" : [ {
          "value" : 1.0,
          "description" : "termFreq=1.0"
        } ]
      }, {
        "value" : 0.30685282,
        "description" : "idf(docFreq=1, maxDocs=1)"
      }, {
        "value" : 0.625,
        "description" : "fieldNorm(doc=0)"
      } ]
    } ]
  }
}
```

(6) 返回字段

默认情况下，返回的每个文档中，Elasticsearch将包括索引名称、类型名称、文档标识符、得分和_source字段。我们可以修改这个行为，通过添加fields参数并指定一个以逗号分隔的字段名称列表。这些字段将在存储字段（如果存在的话）或内部_source字段中检索。默认情况下，字段的fields参数值是_source。一个例子是fields=title。

> 也可以加上_source参数并把值设为false，来禁用_source字段的读取。

(7) 结果排序

通过使用sort参数，可以指定自定义排序。Elasticsearch的默认行为是把返回文档按它们的得分降序排列。如果想有不同的排序，则需要指定sort参数。例如，添加sort=published:desc，文档将按published字段降序排序；添加sort=published:asc，则告诉Elasticsearch把文档按published字段升序排序。

如果指定自定义排序，Elasticsearch将省略计算文档的_score字段。这可能不是你想要的。如果在自定义排序的同时还想保持追踪每个文档的得分，你应该把track_scores=true添加到你的查询。请注意，进行自定义排序时跟踪分数，会使查询稍微慢一点（你可能根本察觉不到），因为需要处理能力来计算得分。

(8) 搜索超时

默认情况下，Elasticsearch没有查询超时，但你可能希望查询在一段时间（比如5秒）后超时。Elasticsearch允许你设置timeout参数。查询将执行到给定的timeout值，在那一刻，收集的结果将返回。把timeout=5s添加到你的查询，就可指定一个5秒的超时。

(9) 查询结果窗口

Elasticsearch允许你指定结果窗口（应返回的结果列表中文件的范围）。有两个参数用来指定结果窗口大小：size和from。size参数默认为10，它定义了返回结果的最大数量。from参数的默认值为0，它指定结果应该从哪个记录开始返回。为了从第11个开始返回5个文档，我们将在查询中添加以下参数：size=5&from=10

(10) 搜索类型

URI查询允许使用search_type参数指定搜索类型，搜索类型默认为query_then_fetch。我们可以使用以下6个值：

- dfs_query_then_fetch
- dfs_query_and_fetch
- query_then_fetch
- query_and_fetch
- count
- scan

3.2节将介绍更多搜索类型的相关知识。

(11) 小写扩展词

一些查询使用查询扩展，比如前缀查询（prefix query），3.9节将讨论这一内容。可以使用

lowercase_expanded_terms属性来定义扩展词是否应该被转为小写。默认该属性为true，意味着扩展词将被小写。

(12) 分析通配符和前缀

默认情况下，通配符查询和前缀查询不会被分析。如果要更改此行为，可以把analyze_ildcard属性设置为true。

1.5.3　Lucene查询语法

我们认为最好大致了解在URI查询里的q参数中可以使用的语法。Elasticsearch中的一些查询，比如正在讨论的这个查询，支持使用Lucene查询解析器语法，这是一种用来构建查询的语言。来看看它并讨论一些基本功能。如果想阅读完整的Lucene查询语法，请访问如下网页：http://lucene.apache.org/core/4_6_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html。

我们传到Lucene的查询被查询解析器分为词（term）和操作符（operator）。先从词开始，你可以区分两种类型的词：单词和短语。例如，为了查询title字段中的book一词，传入如下查询：

```
title:book
```

为了查询title字段中elasticsearch book这个短语，传入如下查询：

```
title:"elasticsearch book"
```

你可能已经注意到，字段的名字在前面，单词或短语在后面。

前面说过，Lucene查询语法支持操作符。例如，操作符+告诉Lucene给定部分必须在文档中匹配。操作符-正相反，查询的这一部分不能出现在文档中。查询中既没有+又没有 – 操作符的部分将被视为可以匹配、但非强制性的查询。所以，如果想找title字段包含book一词但description字段不包含cat一词的文档，传入以下查询：

```
+title:book -description:cat
```

也可以用括号来组合多个词，如下面的查询：

```
title:(crime punishment)
```

还可以使用^操作符接上一个值来助推（boost）[①]一个词，比如以下查询：

```
title:book^4
```

① boost将加强查询对该词的相关性。——译者注

1.6 小结

本章介绍了什么是全文搜索，以及Apache Lucene是如何实现的；熟悉了Elasticsearch的基本概念和它的顶层架构；使用Elasticsearch REST API来索引、更新、检索，最终删除数据；最后，使用简单的URI查询搜索了我们的数据。下一章的重点是建立索引数据。我们将看到Elasticsearch索引的工作原理，主分片和其副本的作用。还会看到Elasticsearch如何处理它不知道的数据，或者说如何创建我们自己的映射，也就是描述索引结构的JSON结构。我们还将学习如何使用批量索引来加快索引过程，可以存储什么额外的信息来帮助实现目标。此外，我们将讨论什么是索引段，什么是段合并以及如何调整段。最后，我们将看到在Elasticsearch中路由是如何工作的，以及在谈到索引路由和查询路由时，有什么选择。

第 2 章 索 引

上一章介绍了关于全文搜索和Elasticsearch的基础知识，也知道了什么是Apache Lucene，还了解到Elasticsearch的安装、标准目录布局及注意事项。我们创建了索引，检索并更新了数据，最后使用简单的URI查询从Elasticsearch获得数据。在本章结束之时，你将学到以下内容：

- Elasticsearch索引；
- 配置索引结构映射，知道可使用的字段类型；
- 使用批量索引加快索引过程；
- 使用附加的内部信息扩展索引结构；
- 理解、设置及控制段合并；
- 理解路由的工作原理，并根据需求设置。

2.1 Elasticsearch 索引

我们已经启动并运行Elasticsearch集群，知道了如何使用Elasticsearch REST API索引、删除和检索数据，如何通过搜索来获取文档。如果你用过SQL数据库，或许会知道，在存入数据前，需要创建用来描述数据的一个结构。尽管Elasticsearch是一个无模式的搜索引擎，可以即时算出数据结构，但我们仍认为由自己控制并定义结构是更好的方法。在接下来的内容中，你将看到如何创建和删除索引。在深入了解可用的API方法之前，先了解一下建立索引的过程。

2.1.1 分片和副本

回忆之前的章节，Elasticsearch索引是由一个或多个分片组成的，每个分片包含了文档集的一部分。而且这些分片也可以有副本，它们是分片的完整副本。在创建索引的过程中，可以规定应创建的分片及副本的数量。也可以忽略这些信息，并使用全局配置文件（elasticsearch.yml）定义的默认值，或Elasticsearch内部实现的默认值。如果我们依赖Elasticsearch的默认值，索引结束时将得到5个分片及1个副本。这意味着什么？简单来说，操作结束时，将有10个Lucene索引分布在集群中。

> 想知道如何通过5个分片和1个副本计算得出有10个Lucene索引？“副本”（replica）这个术语有些误导。它意味着每一个分片都有自己的分片副本（copy），所以实际上有5个分片和5个相应分片副本。

一般而言，同时具有分片和与其相应的副本，意味着建立索引文档时，两者都得修改。这是因为要使分片得到精确的副本，Elasticsearch需将分片的变动通知所有副本。若要读取文件，可以使用分片或者其副本。在具有许多物理节点的系统中，可以把分片和副本放置于不同节点上，从而发挥更多处理能力（如磁盘I/O或CPU）。综上所述，得出结论如下。

- 更多分片使索引能传送到更多服务器，意味着可以处理更多文件，而不会降低性能。
- 更多分片意味着获取特定文档所需的资源量会减少，因为相较于部署更少分片时，存储在单个分片中的文件数量更少。
- 更多分片意味着搜索索引时会面临更多问题，因为必须从更多分片中合并结果，使得查询的聚合阶段需要更多资源。
- 更多副本会增强集群系统的容错性，因为当原始分片不可用时，其副本将替代原始分片发挥作用。只拥有单个副本，集群可能在不丢失数据的情况下遗失分片。当有两个副本时，即使丢失了原始分片及其中一个副本，一切工作仍可以很好地持续下去。
- 更多副本意味着查询吞吐量将会增加，因为执行查询可以使用分片或分片的任一副本。

当然，Elasticsearch中分片和副本的数量之间还有其他关系，稍后将讨论。

那么，应该以什么标准来确定分片和副本的数量？这要视情况而定。我们相信，默认值确实不错，但一个好的测试无法取代。需要注意的是，副本的数量相对没那么重要，因为可以在生成索引后，在生产环境的集群中调整。只要你想，并且有足够的资源，就可以删除和添加副本。但对于分片来说，这样的操作就不可能了。一旦创建好索引，更改分片数量的唯一途径就是创建另一个索引并重新索引数据。

2.1.2　创建索引

在Elasticsearch中创建第一个文档时，没有关心索引的建立，只是使用了如下命令：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New
  version of Elasticsearch released!", "content": "...", "tags":
  ["announce", "elasticsearch", "release"] }'
```

这是可以的。如果这样的索引不存在，Elasticsearch会为我们自动创建索引。还可以通过运行以下命令来创建索引：

```
curl -XPUT http://localhost:9200/blog/
```

我们只是告诉Elasticsearch需要创建名为`blog`的索引。顺利的话，你会从Elasticsearch看到如

下响应：

```
{"acknowledged":true}
```

什么时候需要手动创建索引？有许多情况，比如额外设置时，设置索引结构或分片数目。

1. 修改索引的自动创建

有时你会觉得，自动创建索引并非是一件好事。当你有一个大系统，需要很多流程来将数据输送到Elasticsearch时，索引名称的一个简单拼写错误可能会破坏掉几小时的脚本工作。你可以通过在elasticsearch.yml配置文件中添加以下指令来关闭自动创建索引：

```
action.auto_create_index: false
```

> 需要注意的是，action.auto_create_index比看起来要复杂。我们不但可以把它的值设置成false或ture， 也可以使用索引的名字模式来指定是否在具有给定名字的索引不存在时自动创建。例如在下面的例子中，允许自动创建以a开头的索引，但以an开头的索引则不允许。其他索引也必须手动创建（因为指令中的-*）。
>
> ```
> action.auto_create_index: -an*,+a*,-*
> ```
>
> 注意，模式定义的顺序很重要。Elasticsearch检查这些模式直到第一种匹配的模式，所以，如果你将-an*移动到最后，它将不会被使用，因为指令中含有+a*，所以优先使用+a*。

2. 新创建索引的设定

想设置一些配置选项时，也需要手动创建索引，例如设置分片和副本的数量。来看看下面的例子：

```
curl -XPUT http://localhost:9200/blog/ -d '{
    "settings" : {
        "number_of_shards" : 1,
        "number_of_replicas" : 2
    }
}'
```

上面的命令将创建名为blog的索引，它将有1个分片和2个副本，即共得到3个物理Lucene索引。此外，也可以通过这种方式设置其他值，稍后讨论。

所以，我们已经创建了新的索引。但是有一个问题，我们忘了提供映射来描述索引结构。该怎么做？既然还没有任何数据，那就选择最简单的方法：删除索引。为此，将运行类似于上面的指令，然而不是用HTTP PUT方法，而是使用DELETE。实际指令如下所示：

```
curl -XDELETE http://localhost:9200/posts
```

响应将与我们之前看到的一样，如下所示：

```
{"acknowledged":true}
```

现在我们知道什么是索引，如何创建和删除，我们已经准备好用定义好的映射来创建索引。这部分非常重要，因为数据索引化将会影响搜索过程和文档匹配方式。

2.2 映射配置

如果你习惯用SQL数据库，或许会知道，在存入数据前需要创建模式以描述数据。尽管Elasticsearch是一个无模式的搜索引擎，可以即时算出数据结构，我们仍认为由自己控制并定义结构是更好的方法。在接下来的内容中，你将看到如何创建及删除新的索引，也将了解如何创建映射以满足需求和匹配你的数据结构。

> 注意，我们并不会在本章中阐述现有类型的全部信息。Elasticsearch的嵌套类型、主从关系处理、存储地理点以及搜索等特性，将在接下来的章节说明。

2.2.1 类型确定机制

在开始描述如何手动创建映射之前，我们想说明一件事。Elasticsearch可以通过定义文档的JSON来猜测文档结构。在JSON中，字符串用引号括起来，布尔值使用特定的词语定义，数值则是一些数字。这是一个简单的技巧，但通常有效。举例说明，请看以下文档：

```
{
  "field1": 10,
  "field2": "10"
}
```

上面的文档有两个字段。`field1`将被确定为数字（number，准确地说是`long`类型），但`field2`被确定为字符串，因为它用引号括起来。当然，这是我们所需要的行为，但有时数据源会省略掉数据类型的相关信息，一切都以字符串的形式呈现。解决方案是在映射定义文件中把`numeric_detection`属性设置为`true`，以开启更积极的文本检测。比如，可以在索引的创建过程中执行以下命令：

```
curl -XPUT http://localhost:9200/blog/?pretty -d '{
  "mappings" : {
    "article": {
      "numeric_detection" : true
    }
  }
}'
```

可惜，如果我们想要猜中布尔类型，问题仍然存在。我们不能从文本中强制推测出布尔类型。

在这种情况下，当无法改变源格式时，只能在映射定义中直接定义字段。

造成麻烦的另一个类型是基于日期类型的字段。Elasticsearch设法猜测被提供的时间戳或与日期格式匹配的字符串。可以使用dynamic_date_formats属性定义可被识别的日期格式列表，该属性允许指定一个格式的数组。看一下创建索引和类型的命令：

```
curl -XPUT 'http://localhost:9200/blog/' -d '{
  "mappings" : {
    "article" : {
      "dynamic_date_formats" : ["yyyy-MM-dd hh:mm"]
    }
  }
}'
```

上述命令可以创建名为blog的索引，其中包含一个名为article的类型。我们还使用单一日期格式的dynamic_date_formats属性，这样，对于与此格式匹配的字段，Elasticsearch将使用date核心类型（请参阅本章的“核心类型”部分了解更多关于字段类型的信息）。Elasticsearch使用joda-time库定义日期格式，所以如果有兴趣了解更多详情，请访问http://joda-time.sourceforge.net/api-release/org/joda/time/format/DateTimeFormat.html。

> 记住，dynamic_date_format属性可接受一个数组。这意味着，我们可以同时处理多种日期格式。

禁用字段类型猜测

想象以下情况。首先，索引一个数字，一个整数。Elasticsearch会猜测其类型，并设置类型为整数型（integer）或长整型（long）（请参阅本章的“核心类型”部分了解更多关于字段类型的信息）。如果索引另一个文档，它在同一个字段中存储的是浮点数，会发生什么？Elasticsearch将会删除小数部分并存储剩余整数。关闭它的另一个原因在于我们不希望在现有索引中添加新字段：那些在应用程序开发过程中未知的字段。

要关闭自动添加字段，可以把dynamic属性设置为false。把dynamic属性添加为类型的属性。例如，要在blog索引中为article类型关闭自动字段类型猜测，命令如下所示：

```
curl -XPUT 'http://localhost:9200/blog/' -d '{
  "mappings" : {
    "article" : {
      "dynamic" : "false",
      "properties" : {
        "id" : { "type" : "string" },
        "content" : { "type" : "string" },
        "author" : { "type" : "string" }
      }
    }
  }
}'
```

使用上面的命令创建blog索引后，在properties部分（下一节讨论）未提到的字段会被Elasticsearch忽略。如此，除id、content和author以外的任何字段都将被忽略。当然，这只发生在blog索引的article类型中。

2.2.2 索引结构映射

模式映射（schema mapping，或简称映射）用于定义索引结构。你可能还记得，每个索引可以有多种类型，但现在会为了简单起见我们只专注一个类型。假设想创建一个保存博客帖子数据的posts索引。它可以具有以下结构：

- 唯一标识符；
- 名称；
- 发布日期；
- 内容。

在Elasticsearch中，映射在文件中以JSON对象传送。所以，创建一个映射文件来匹配上述需求，称之为posts.json。其内容如下：

```
{
  "mappings": {
    "post": {
      "properties": {
        "id": {"type":"long", "store":"yes",
        "precision_step":"0" },
        "name": {"type":"string", "store":"yes",
        "index":"analyzed" },
        "published": {"type":"date", "store":"yes",
        "precision_step":"0" },
        "contents": {"type":"string", "store":"no",
        "index":"analyzed" }
      }
    }
  }
}
```

为使用上述文件创建posts索引，运行以下命令（假设我们的映射存储在posts.json文件中）：

```
curl -XPOST 'http://localhost:9200/posts' -d @posts.json
```

> 注意，你可以把映射存储成你想要的任何文件名。

同样，顺利的话，可以看到如下响应：

```
{"acknowledged":true}
```

现在，有了索引结构，可以索引我们的数据。休息一下，先来讨论posts.json文件的内容。

1. 类型定义

可以看到，posts.json文件内容是个JSON对象，因此它被大括号括起来（了解更多关于JSON的信息，请参看http://www.json.org/）。上述文件中的所有类型定义都嵌套在mappings对象中。你可以在mappings的JSON对象内定义多种类型。在在我们的例子中，只有单一的post类型。但如果还要包括user类型，文件将如下所示：

```
{
  "mappings": {
    "post": {
      "properties": {
        "id": { "type":"long", "store":"yes",
        "precision_step":"0" },
        "name": { "type":"string", "store":"yes",
        "index":"analyzed" },
        "published": { "type":"date", "store":"yes",
        "precision_step":"0" },
        "contents": { "type":"string", "store":"no",
        "index":"analyzed" }
      }
    },
    "user": {
      "properties": {
        "id": { "type":"long", "store":"yes",
        "precision_step":"0" },
        "name": { "type":"string", "store":"yes",
        "index":"analyzed" }
      }
    }
  }
}
```

2. 字段

每种类型由一组属性定义，也就是定义在properties对象中的字段。先集中在单个字段上，如contents字段，其完整定义如下所示：

```
"contents": { "type":"string", "store":"yes", "index":"analyzed" }
```

它由字段的名称开始，上述例子中是contents。名称后面，使用一个对象指定该字段的行为。我们所用字段的类型有特定的属性，下一节将讨论它们。当然，如果单一类型拥有多个字段（这是常见情况），记得用逗号将它们隔开。

3. 核心类型

每个字段类型可以指定为Elasticsearch提供的一个特定核心类型。Elasticsearch有以下核心类型。

- string：字符串；

- number：数字；
- date：日期；
- boolean：布尔型；
- binary：二进制。

现在来讨论Elasticsearch中可用的每个核心类型，以及它们用来定义行为的属性。

(1) 公共属性

在继续描述所有核心类型之前，先讨论一些可用来描述所有类型（二进制除外）的公共属性。

- index_name：该属性定义将存储在索引中的字段名称。若未定义，字段将以对象的名字来命名。
- index：可设置值为analyzed和no。另外，对基于字符串的字段，也可以设置为not_analyzed。如果设置为analyzed，该字段将被编入索引以供搜索。如果设置为no，将无法搜索该字段。默认值为analyzed。在基于字符串的字段中，还有一个额外的选项not_analyzed。此设置意味着字段将不经分析而编入索引，使用原始值被编入索引，在搜索的过程中必须全部匹配。索引属性设置为no将使include_in_all属性失效。
- store：这个属性的值可以是yes或no，指定了该字段的原始值是否被写入索引中。默认值设置为no，这意味着在结果中不能返回该字段（然而，如果你使用_source字段，即使没有存储也可返回这个值），但是如果该值编入索引，仍可以基于它来搜索数据。
- boost：该属性的默认值是1。基本上，它定义了在文档中该字段的重要性。boost的值越高，字段中值的重要性也越高。
- null_value：如果该字段并非索引文档的一部分，此属性指定应写入索引的值。默认的行为是忽略该字段。
- copy_to：此属性指定一个字段，字段的所有值都将复制到该指定字段。
- include_in_all：此属性指定该字段是否应包括在_all字段中。默认情况下，如果使用_all字段，所有字段都会包括在其中。2.4节将更详细地介绍_all字段。

(2) 字符串

字符串是最基本的文本类型，我们能够用它存储一个或多个字符。字符串字段的示例定义如下所示：

```
"contents" : { "type" : "string", "store" : "no", "index" :
  "analyzed" }
```

除了公共属性，基于字符串的字段还可以使用以下属性。

- term_vector：此属性的值可以设置为no（默认值）、yes、with_offsets、with_positions和with_positions_offsets。它定义是否要计算该字段的Lucene词向量（term vector）。如果你使用高亮，那就需要计算这个词向量。

- omit_norms：该属性可以设置为true或false。对于经过分析的字符串字段，默认值为false，而对于未经分析但已编入索引的字符串字段，默认值设置为true。当属性为true时，它会禁用Lucene对该字段的加权基准计算（norms calculation），这样就无法使用索引期间的加权，从而可以为只用于过滤器中的字段节省内存（在计算所述文件的得分时不会被考虑在内）。
- analyzer：该属性定义用于索引和搜索的分析器名称。它默认为全局定义的分析器名称。
- index_analyzer：该属性定义了用于建立索引的分析器名称。
- search_analyzer：该属性定义了的分析器，用于处理发送到特定字段的那部分查询字符串。
- norms.enabled：此属性指定是否为字段加载加权基准（norms）。默认情况下，为已分析字段设置为true（这意味着字段可加载加权基准），而未经分析字段则设置为false。
- norms.loading：该属性可设置eager和lazy。第一个属性值表示此字段总是载入加权基准。第二个属性值是指只在需要时才载入。
- position_offset_gap：此属性的默认值为0，它指定索引中在不同实例中具有相同名称的字段的差距。若想让基于位置的查询（如短语查询）只与一个字段实例相匹配，可将该属性值设为较高值。
- index_options：该属性定义了信息列表（postings list）的索引选项（2.2.4节将详细讨论）。可能的值是docs（仅对文档编号建立索引），freqs（对文档编号和词频建立索引），positions（对文档编号、词频和它们的位置建立索引），offsets（对文档编号、词频、它们的位置和偏移量建立索引）。对于经分析的字段，此属性的默认值是positions，对于未经分析的字段，默认值为docs。
- ignore_above：该属性定义字段中字符的最大值。当字段的长度高于指定值时，分析器会将其忽略。

(3) 数值

这一核心类型汇集了所有适用的数值字段类型。Elasticsearch中可使用以下类型（使用type属性指定）。

- byte：定义字节值，例如1。
- short：定义短整型值，例如12。
- integer：定义整型值，例如134。
- long：定义长整型值，例如123456789。
- float：定义浮点值，例如12.23。
- double：定义双精度值，例如123.45。

> 你可以在如下链接中了解更多所提到的Java类型：http://docs.oracle.com/avase/tutorial/java/nutsandbolts/datatypes.html。

数值类型字段的定义如下所示：

```
"price" : { "type" : "float", "store" : "yes", "precision_step" : "4" }
```

除了公共属性，以下属性也适用于数值字段。

- precision_step：此属性指定为某个字段中每个值生成的词条数。值越低，产生的词条数越高。对于每个值的词条数更高的字段，范围查询（range query）会更快，但索引会稍微大点，默认值为4。
- ignore_malformed：此属性值可以设为true或false。默认值是false。若要忽略格式错误的值，则应设置属性值为true。

(4) 布尔值

布尔值核心类型是专为索引布尔值（true或false）设计的。基于布尔值类型的字段定义如下所示：

```
"allowed" : { "type" : "boolean", "store": "yes" }
```

(5) 二进制

二进制字段是存储在索引中的二进制数据的Base64表示，可用来存储以二进制形式正常写入的数据，例如图像。基于此类型的字段在默认情况下只被存储，而不索引，因此只能提取，但无法对其执行搜索操作。二进制类型只支持index_name属性。基于binary字段的字段定义如下所示：

```
"image" : { "type" : "binary" }
```

(6) 日期

日期核心类型被设计用于日期的索引。它遵循一个特定的、可改变的格式，并默认使用UTC保存。

能被Elasticsearch理解的默认日期格式是相当普遍的，它允许指定日期，也可指定时间，例如，2012-12-24T12:10:22。基于日期类型的字段的示例定义如下所示：

```
"published" : { "type" : "date", "store" : "yes", "format" :
  "YYYY-mm-dd" }
```

使用上述字段的示例文档如下所示：

```
{
  "name" : "Sample document",
  "published" : "2012-12-22"
}
```

除了公共属性，日期类型的字段还可以设置以下属性。

- format：此属性指定日期的格式。默认值为dateOptionalTime。对于格式的完整列表，请访问 http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/mapping-date-ormat.html。
- precision_step：此属性指定在该字段中的每个值生成的词条数。该值越低，产生的词条数越高，从而范围查询的速度越快（但索引大小增加）。默认值是4。
- ignore_malformed：此属性值可以设为true或false，默认值是false。若要忽略格式错误的值，则应设置属性值为true。

4. 多字段

有时候你希望两个字段中有相同的字段值，例如，一个字段用于搜索，一个字段用于排序；或一个经语言分析器分析，一个只基于空白字符来分析。Elasticsearch允许加入多字段对象来拓展字段定义，从而解决这个需求。它允许把几个核心类型映射到单个字段，并逐个分析。例如，想计算切面并在name字段中搜索，可以定义以下字段：

```
"name": {
  "type": "string",
  "fields": {
    "facet": { "type" : "string", "index": "not_analyzed" }
  }
}
```

上述定义将创建两个字段：我们将第一个字段称为name，第二个称为name.facet。当然，你不必在索引的过程中指定两个独立字段，指定一个name字段就足够了。Elasticsearch会处理余下的工作，将该字段的数值复制到多字段定义的所有字段。

5. IP地址类型

Elasticsearch添加了IP字段类型，以数字形式简化IPv4地址的使用。此字段类型可以帮搜索作为IP地址索引的数据、对这些数据排序，并使用IP值做范围查询。

基于IP地址类型的字段示例定义如下所示：

```
"address" : { "type" : "ip", "store" : "yes" }
```

除公共属性外，IP地址类型的字段还可以设置precision_step属性。该属性指定了字段中的每个值生成的词条数。值越低，词条数越高。对于每个值的词条数更高的字段，范围查询会更快，但索引会稍微大点，默认值为4。

使用上述字段的示例文档如下所示：

```
{
  "name" : "Tom PC",
  "address" : "192.168.2.123"
}
```

6. token_count类型

token_count字段类型允许存储有关索引的字数信息，而不是存储及检索该字段的文本。它接受与number类型相同的配置选项，此外，还可以通过analyzer属性来指定分析器。

基于token_count类型的字段示例定义如下：

```
"address_count" : { "type" : "token_count", "store" : "yes" }
```

7. 使用分析器

正如我们提到的那样，对于字符串类型的字段，可以指定Elasticsearch应该使用哪个分析器。回想第1章的内容，分析器是一个用于分析数据或以我们想要的方式查询数据的工具。例如，用空格和小写字符把单词隔开时，不必担心用户发送的单词是小写还是大写。Elasticsearch使我们能够在索引和查询时使用不同的分析器，并且可以在搜索过程的每个阶段选择处理数据的方式。使用分析器时，只需在指定字段的正确属性上设置它的名字，就这么简单。

(1) 开箱即用的分析器

Elasticsearch允许我们使用众多默认定义的分析器中的一种。如下分析器可以开箱即用。

- standard：方便大多数欧洲语言的标准分析器（关于参数的完整列表，请参阅http://www.lasticsearch.org/guide/en/elasticsearch/reference/current/analysis-standard-analyzer.html）。
- simple：这个分析器基于非字母字符来分离所提供的值，并将其转换为小写形式。
- whitespace：这个分析器基于空格字符来分离所提供的值。
- stop：这个分析器类似于simple分析器，但除了simple分析器的功能，它还能基于所提供的停用词（stop word）过滤数据（参数的完整列表，请参阅http://www.elasticsearch.rg/guide/en/elasticsearch/reference/current/analysis-stop-analyzer.html）。
- keyword：这是一个非常简单的分析器，只传入提供的值。你可以通过指定字段为not_analyzed来达到相同的目的。
- pattern：这个分析器通过使用正则表达式灵活地分离文本（参数的完整列表，请参阅http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-pattern-analyzer.html）。
- language：这个分析器旨在特定的语言环境下工作。该分析器所支持语言的完整列表可参考http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer. tml。

- snowball：这个分析器类似于standard分析器，但提供了词干提取算法（stemming algorithm，参数的完整列表请参阅 http://www.elasticsearch.org/guide/en/elasticsearch/eference/current/analysis-snowball-analyzer.html）。

> 词干提取（stemming）是还原屈折词和派生词至其基本形式的过程。这种方法缩减了字数，例如，对cars和car这两个单词，词干提取器（stemmer，词干提取算法的一种实现）产生一个词干car。索引完成后，在查询任一单词时，包含这些词的文档都会被匹配。如果不做词干提取，只有用cars查询时，包含“cars”的文档才会被匹配。

(2) 定义自己的分析器

除了前面提到的分析器，Elasticsearch还允许我们定义新的分析器，而无需编写Java代码。为此，需要在映射文件中加入settings节，它包含Elasticsearch创建索引时所需要的有用信息。下面演示了如何自定义settings节：

```
"settings" : {
  "index" : {
    "analysis": {
      "analyzer": {
        "en": {
          "tokenizer": "standard",
          "filter": [
            "asciifolding",
            "lowercase",
            "ourEnglishFilter"
          ]
        }
      },
      "filter": {
        "ourEnglishFilter": {
          "type": "kstem"
        }
      }
    }
  }
}
```

我们指定一个新的名为en的分析器。每个分析器由一个分词器和多个过滤器构成。默认过滤器和分词器的完整列表可以参阅 http://www.elasticsearch.org/guide/en/elasticsearch/reference/urrent/analysis.html。我们的en分析器包括standard分词器和三个过滤器：默认情况下可用的asciifolding和lowercase，以及一个自定义的ourEnglishFilter。

要想定义过滤器，需要提供它的名称、类型以及该过滤器类型需要的任意数量的附加参数。Elasticsearch 中可用过滤器类型的完整列表可以参考 http://www.elasticsearch.org/guide/en/

elasticsearch/reference/current/analysis.html。此列表不断更新，所以这里不予讨论。

所以，定义了分析器的映射文件最终将如下所示：

```
{
  "settings" : {
    "index" : {
      "analysis": {
        "analyzer": {
          "en": {
            "tokenizer": "standard",
            "filter": [
              "asciifolding",
              "lowercase",
              "ourEnglishFilter"
            ]
          }
        },
        "filter": {
          "ourEnglishFilter": {
            "type": "kstem"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id": { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name": { "type" : "string", "store" : "yes", "index" :
        "analyzed", "analyzer": "en" }
      }
    }
  }
}
```

可以通过Analyze API了解分析器的工作情况（http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/indices-analyze.html）。例如下面的命令：

```
curl -XGET 'localhost:9200/posts/_analyze?pretty&field=post.name' -d
'robots cars'
```

上面的命令要求Elasticsearch展示为`post`类型和它的`name`字段定义的分析器对指定短语（robots cars）的分析内容，得到的响应如下：

```
{
  "tokens" : [ {
    "token" : "robot",
    "start_offset" : 0,
    "end_offset" : 6,
```

```
      "type" : "<ALPHANUM>",
      "position" : 1
    }, {
      "token" : "car",
      "start_offset" : 7,
      "end_offset" : 11,
      "type" : "<ALPHANUM>",
      "position" : 2
    } ]
}
```

可以看到，短语robots cars被分离成两个词条。另外，单词robots更改成robot，cars更改成car。

(3) 分析器字段

可以通过分析器字段（_analyzer）指定一个字段，该字段的值将作为字段所属文档的分析器名称。试想一下，你有个软件检测写入文档的语言，并在文档的language字段存储相关信息。此外，你想使用这些信息来选择合适的分析器。为此，只需添加以下几行代码到你的映射文件：

```
"_analyzer" : {
  "path" : "language"
}
```

包含上述信息的映射文件如下所示：

```
{
  "mappings" : {
    "post" : {
      "_analyzer" : {
        "path" : "language"
      },
      "properties" : {
        "id": { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name": { "type" : "string", "store" : "yes",
        "index" : "analyzed" },
        "language": { "type" : "string", "store" : "yes",
        "index" : "not_analyzed"}
      }
    }
  }
}
```

需要注意的是，应该定义一个与language字段中提供的值一样的分析器，否则索引将会失败。

(4) 默认分析器

关于分析器，还有一点需要说明，即在没有定义分析器的情况下，应指定在默认情况下使用的分析器。这与在映射文件中的setting部分配置自定义分析器的方式相同，但应使用default关键字来命名，而不是为分析器指定一个自定义名称。因此，为了把前面定义的分析器作为默认分析器，可以将en分析器修改为下面这样：

```
{
  "settings" : {
    "index" : {
      "analysis": {
        "analyzer": {
          "default": {
            "tokenizer": "standard",
            "filter": [
              "asciifolding",
              "lowercase",
              "ourEnglishFilter"
            ]
          }
        },
        "filter": {
          "ourEnglishFilter": {
            "type": "kstem"
          }
        }
      }
    }
  }
}
```

2.2.3 不同的相似度模型

2012年发布Apache Lucene 4.0后，该全文检索库的所有用户便可以修改默认的基于TF/IDF的算法（第1章中提到过）。但这不是唯一的变化，Lucene 4.0还附加了相似度模型（similarity model），允许在文档中使用不同的评分公式。

1. 设定每个字段的相似度模型

自Elasticsearch 0.90版本开始，就可以为映射文件中的每个字段设置不同的相似度模型。例如，使用如下简单的映射来索引`blog posts`：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
          "index" : "analyzed" }
      }
    }
  }
}
```

为此,可在name字段和contents字段中使用BM25相似度模型。这需要扩展我们的字段定义,并添加similarity属性以及所选择的similarity名称的值。修改映射如下:

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
          "index" : "analyzed", "similarity" : "BM25" },
        "contents" : { "type" : "string", "store" : "no",
          "index" : "analyzed", "similarity" : "BM25" }
      }
    }
  }
}
```

仅此而已,无需再添加别的。做出上述修改后,Lucene Apache将为字段name和contents使用BM25相似度模型计算得分因子。

2. 可用的相似度模型

现有如下三种相似度模型。

- Okapi BM25模型:这种相似度模型基于概率模型,概率模型估算根据指定查询找到指定文档的概率。为了在Elasticsearch中使用这种相似度模型,需要将BM25作为名称。据说Okapi BM25相似度模型在处理简短的文本文档时表现最佳,在这种文档中词条的重复严重有损整体文档的得分。要使用此相似度模型,需要设置字段的similarity属性为BM25。这种相似度模型在定义后立即可用,不需要设置附加属性。
- 随机性偏差(divergence from randomness)模型:这种相似度模型基于具有相同名称的概率模型。为了在Elasticsearch中使用这种相似度模型,需要使用DFR作为名称。随机性偏差模型在处理类自然语言文本时表现良好。
- 信息基础(information-based)模型:这是新推出的最后一个相似度模型,与随机性偏差模型非常相似。为了在Elasticsearch中使用这种相似度模型,需要使用IB作为名称。类似于DFR相似度模型,信息基础模型在处理类自然语言文本时表现良好。

(1) 配置DFR相似度模型

在DFR相似度模型中,可以配置basic_model属性(可设置为be、d、g、if、in或者ine),after_effect属性(可设置为no、b或l),normalization属性(可设置为no、h1、h2、h3或z)。如果选择no以外的值作为normalization属性值,需要设置范式化因子(normalization factor)。所选择的normalization值如果是h1,则设置normalization.h1.c(浮点值);如果是h2,则设置normalization.h2.c(浮点值);如果是h3,则设置normalization.h3.c

（浮点值）；如果是z，则设置`normalization.z.z`（浮点值）。例如，下面是相似度配置的一个示例（把它放到映射文件的settings部分中）：

```
"similarity" : {
  "esserverbook_dfr_similarity" : {
    "type" : "DFR",
    "basic_model" : "g",
    "after_effect" : "l",
    "normalization" : "h2",
    "normalization.h2.c" : "2.0"
  }
}
```

(2) 配置IB相似度模型

在IB相似度模型中，有以下参数可供配置：`distribution`属性（可设置为`ll`或`spl`）和`lambda`属性（可设置为`df`或`tff`）。此外，跟DFR相似度模型一样，可以选择范式化因子，这里不再赘述。下面是一个配置IB相似度模型的例子（把它放到配置文件的`settings`部分）：

```
"similarity" : {
  "esserverbook_ib_similarity" : {
    "type" : "IB",
    "distribution" : "ll",
    "lambda" : "df",
    "normalization" : "z",
    "normalization.z.z" : "0.25"
  }
}
```

> 相似度模型是一个相当复杂的话题，需要一整章来正确地描述它。如果你有兴趣，请参阅*Mastering ElasticSearch*，或到http://elasticsearchserverbook.com/elasticsearch-0-90-similarities/阅读更多细节。

2.2.4 信息格式

Apache Lucene 4.0的显著变化之一是，可以改变索引文件写入的方式。Elasticsearch利用了此功能，可以为每个字段指定信息格式。有时你需要改变字段被索引的方式以提高性能，比如为了使主键查找更快。

Elasticsearch中的信息格式如下所示。

- `default`：没有明确定义格式时，此默认信息格式将被使用。它提供了实时的对存储字段和词向量的压缩。如果你想知道压缩的内容，请参阅http://solr.pl/en/2012/11/19/solr-4-1-stored-fields-compression/。

- ❑ pulsing：此信息格式将高基数字段（high cardinality field）[①]的信息列表编码为词条矩阵，这让Lucene检索文档时可以少执行一个搜索。对高基数字段使用此信息格式可以加快此字段的查询速度。
- ❑ direct：此信息格式可在读操作过程中将词条加载到矩阵中。这些矩阵未经压缩保存在内存中。这种信息格式可以提升常用字段的性能，但是使用时应谨慎，因为它属于内存密集型，词条和信息矩阵都存储于内存中。请记住，因为所有的词条都存储在比特组里，所以每个段需要多达2.1 GB的内存。
- ❑ memory：此信息格式正如它的名字所示，将所有的数据写入磁盘，但需要用到一个名为FST（Finite State Transducer，有限状态传感器）的结构读取词条和信息列表到内存中。你可以在 http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html 上Mike McCandless的帖子中了解更多关于这个结构的知识。由于数据存储于内存中，这个信息格式会提升常用词条的性能。
- ❑ bloom_default：默认信息格式的扩展，增加了把布隆过滤器（bloom filter）写入磁盘的功能。在读取过程中，布隆过滤器被读取并存入内存，以便非常快速地检查给定的值是否存在。该信息格式对于高基数字段相当有效，比如主键字段。如果你想知道更多关于布隆过滤器的知识，请参阅http://en.wikipedia.org/wiki/Bloom_filter。此信息格式除了默认格式的功能，还使用了布隆过滤器。
- ❑ bloom_pulsing：这是pulsing信息格式的扩展，除了pulsing格式的功能，还使用了布隆过滤器。

配置信息格式

信息格式可在每个字段上设置，就像type或name。为了把字段配置成默认格式以外的格式，需要添加一个名为postings_format的属性，将所选择信息格式的名字作为值。如果想在id字段使用pulsing信息格式，映射将如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0", "postings_format" : "pulsing" },
        "name" : { "type" : "string", "store" : "yes",
          "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
          "index" : "analyzed" }
      }
    }
  }
}
```

① 基数越高，字段的重复值就越少，可选性越高。——译者注

2.2.5 文档值

文档值将是本节讨论的最后一个字段属性。文档值格式是在Lucene 4.0中引入的另一个新功能。它允许定义一个给定字段的值被写入一个具有较高内存效率的列式结构，以便进行高效的排序和切面搜索。使用了文档值的字段将有专属的字段数据缓存实例，无需像标准字段一样倒排（以避免像第1章所描述的方法一样存储）。因此，它使索引刷新操作速度更快，让你可以在磁盘上存储字段数据，从而节省堆内存的使用。

1. 配置文档值

我们扩展一下posts索引示例，增加一个votes字段。假设新添加的字段包含指定文章的得票数量，并且我们希望对它排序。因为需要排序，所以它很适合使用文档值。为了在给定字段上使用文档值，需要将doc_values_format属性添加到其定义中并指定其格式。映射将如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
          "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
          "index" : "analyzed" },
        "votes" : { "type" : "integer",
          "doc_values_format" : "memory" }
      }
    }
  }
}
```

如你所见，定义非常简单。来看看对于doc_values_format属性，有哪些可以设置的值。

2. 文档值格式

目前有以下三种可用的doc_values_format属性值。

- default：当未指定任何格式时，使用此默认格式。此格式使用少量内存而且性能良好。
- disk：此文档值格式将数据存入磁盘，几乎无需内存。然而，在使用这种数据结构执行切面和排序等操作时，性能略有降低。需要执行切面或排序操作，而又苦恼于内存空间时，可使用这种格式。
- memory：此文档值格式将数据存入内存。这种格式中，切面或排序的功能与标准倒排索引字段的功能不相上下。由于这种数据结构存储于内存中，索引的刷新速度更快，而这对快速更改索引及缩短索引更新频率很有帮助。

2.3　批量索引以提高索引速度

在第1章中，我们学习了如何将特定文档添加索引至Elasticsearch。现在来学习如何以更方便有效的方式索引多个文档，而不是逐个索引。

2.3.1　为批量索引准备数据

Elasticsearch可以合并多个请求至单个包中，而这些包可以单个请求的形式传送。如此，可以将如下操作结合起来：

- 在索引中增加或更换现有文档（`index`）；
- 从索引中移除文档（`delete`）；
- 当索引中不存在其他文档定义时，在索引中增加新文档（`create`）。

为了获得较高的处理效率，选择这样的请求格式。它假定，请求的每一行包含描述操作说明的JSON对象，第二行为JSON对象本身。可以把第一行视为信息行，第二类为数据行。唯一的例外是`delete`操作，它只包含信息行。来看看下面的例子：

```
{ "index": { "_index": "addr", "_type": "contact", "_id": 1 }}
{ "name": "Fyodor Dostoevsky", "country": "RU" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 }}
{ "name": "Erich Maria Remarque", "country": "DE" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 }}
{ "name": "Joseph Heller", "country": "US" }
{ "delete": { "_index": "addr", "_type": "contact", "_id": 4 }}
{ "delete": { "_index": "addr", "_type": "contact", "_id": 1 }}
```

重要的是，每一个文档或操作说明放置在一行中（以换行符结束）。这意味着无法美化文档格式。批量索引文件的大小存在限制，它被设定为100 MB，在Elasticsearch配置文件中可以通过`http.max_content_length`属性来改变。这避免了请求过大时可能存在的请求超时及内存问题。

> 需要注意的是，在一个批量索引文档中，可以将数据载入多个索引中，文档也可以有不同的类型。

2.3.2　索引数据

为了执行批量请求，Elasticsearch提供了`_bulk`端点，形式可以是`/_bulk`，也可以是`/index_name/_bulk`，甚至是`/index_name/type_name/_bulk`。第二种和第三种形式定义了索引名称和类型名称的默认值。可以在请求的信息行中省略这些属性，Elasticsearch将使用默认值。

假设已经在documents.json文件中存储了数据，可以运行下面的命令将这些数据发送到Elasticsearch：

```
curl -XPOST 'localhost:9200/_bulk?pretty' --data-binary
  @documents.json
```

没必要设置?pretty参数。之前使用这个参数仅仅是为了方便分析命令的响应。在这个例子中，我们在curl中使用--data-binary参数，而不是使用-d，这很重要。这是因为标准的-d参数忽略换行符，正如前面所说的那样，换行符在解析Elasticsearch的批量请求内容时很重要。现在，来看看由Elasticsearch返回的响应：


```
{
  "took" : 139,
  "errors" : true,
  "items" : [ {
    "index" : {
      "_index" : "addr",
      "_type" : "contact",
      "_id" : "1",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "create" : {
      "_index" : "addr",
      "_type" : "contact",
      "_id" : "2",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "create" : {
      "_index" : "addr",
      "_type" : "contact",
      "_id" : "2",
      "status" : 409,
      "error" : "DocumentAlreadyExistsException[[addr][3]
         [contact][2]: document already exists]"
    }
  }, {
    "delete" : {
      "_index" : "addr",
      "_type" : "contact",
      "_id" : "4",
      "_version" : 1,
      "status" : 404,
      "found" : false
    }
  }, {
    "delete" : {
      "_index" : "addr",
      "_type" : "contact",
      "_id" : "1",
      "_version" : 2,
      "status" : 200,
```

```
        "found" : true
      }
    }]
  }
```

正如我们看到的，每一个结果是items数组的一部分。简要对比这些结果和输入的数据：前两个命令index和create执行得很顺利，而第三个操作失败了，因为想用一个已经存在于索引中的标识符来创建一条记录。接下来的两个删除操作都成功了。注意，其中第一个操作试图删除一个不存在的文档。可以看到，这对Elasticsearch来说不是问题。Elasticsearch返回有关每个操作的信息，因此对于大批量请求，响应也是巨大的。

2.3.3　更快的批量请求

批量操作的速度很快，但如果你想知道是否存在更有效、更快的索引方法，可以看看*用户数据报协议*（User Datagram Protocol，UDP）的批量操作。请注意，使用UDP并不能保证数据在与Elasticsearch服务器通信的过程中不会丢失。因此，只有在性能至关重要且比精确度还重要，并且要索引全部文档时，才考虑使用它。

2.4　用附加的内部信息扩展索引结构

除了保存数据的字段之外，还可以与文档一道存储附加的信息。本书已经讨论过各种不同的映射选项及可用的数据类型。现在我们打算更详细地讨论一些并非每天都使用到的Elasticsearch功能，这些功能可以更方便地处理数据。

> 以下讨论的字段类型应定义在适当的类型级别上，它们不是索引范围的类型。

2.4.1　标识符字段

你可能还记得，Elasticsearch索引的每个文档都有自己的标识符和类型。在Elasticsearch中，文档存在两种内部标识符。

第一个是_uid字段，它是索引中文档的唯一标识符，由该文档的标识符和文档类型构成。这基本意味着，不同类型的文档编入到相同的索引时可以具有相同的文档标识符，而Elasticsearch仍能够区分它们。此字段不需要任何额外的设置，它总是被索引，但知道它的存在是有好处的。

持有标识符的第二个字段是_id字段。此字段存储着索引时设置的实际标识符。为了使_id字段能够被索引（或存储，如果需要的话），需要在映射文件中像设置任何其他属性一样添加_id字段的定义（但是，如前面所说，应添加到类型定义的主体中）。所以，book类型的示例定义如下：

```
{
  "book" : {
    "_id" : {
      "index": "not_analyzed",
      "store" : "no"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

可以看到，上述例子在代码中指明希望_id字段不经分析但要编入索引，而且不希望存储。

除了在索引时间指定标识符，也可从指定我们希望从索引文档的一个字段中获取标识符（由于需要额外的解析，速度会稍慢些）。为此，需要设定path属性，并将它的值设置为一个字段名称，该字段的值将作为标识符。如果我们的索引中含有book_id字段，并打算使用它作为_id字段的值，将上述映射文件的内容更改为如下内容：

```
{
  "book" : {
    "_id" : {
      "path": "book_id"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

最后需要记得一件事：当禁用_id字段时，所有需要文档唯一标识符的功能都能继续工作，因为它们将用_uid字段作为代替。

2.4.2 _type字段

之前说过，在Elasticsearch中每个文档至少需要由它的标识符和类型来描述。默认情况下，文档的类型会编入索引，但不会被分析并且不存储。如果想存储这个字段，可按如下方式修改映射文件内容：

```
{
  "book" : {
    "_type" : {
      "store" : "yes"
    },
    "properties" : {
```

```
      .
      .
      .
    }
  }
}
```

也可改成不索引_type字段，但是那样的话，一些查询（例如词条查询）和过滤器将失效。

2.4.3 _all字段

Elasticsearch使用_all字段存储其他字段中的数据以便于搜索。当要执行简单的搜索功能，搜索所有数据（或复制到_all字段的所有字段），但又不想去考虑字段名称之类的事情时，这个字段很有用。默认情况下，_all字段是启用的，包含了索引中所有字段的所有数据。然而，这一字段使得索引有点大，且这并不总是必要的。可以完全禁用_all字段，或排除某些字段。为了在_all字段不包括某个特定字段，我们将使用本章前面讨论的include_in_all属性。要完全关闭_all字段功能，可修改映射文件，如下所示：

```
{
  "book" : {
    "_all" : {
      "enabled" : "false"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

除了enabled属性，_all字段还支持以下属性：

- store
- term_vector
- analyzer
- index_analyzer
- search_analyzer

有关上述属性的信息，请参阅2.2节。

2.4.4 _source字段

_source字段可以在生成索引过程中存储发送到Elasticsearch的原始JSON文档。默认情况下，_source字段会被开启，因为部分Elasticsearch功能依赖于这个字段（如局部更新功能）。除此之

外，当某字段没有存储时，_source字段可用作高亮功能的数据源。但如果不需要这样的功能，可禁用_source字段避免存储开销。为此，需设置_source对象的enabled属性值为false，如下所示：

```
{
  "book" : {
    "_source" : {
      "enabled" : false
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

排除字段和包含字段

还可以告诉Elasticsearch我们希望从_source字段中排除哪些字段，包含哪些字段。可以通过在_source字段定义中添加includes或excludes属性来实现。如果想从_source中排除author路径下的所有字段，映射将如下所示：

```
{
  "book" : {
    "_source" : {
      "excludes" : [ "author.*" ]
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

2.4.5 _index字段

Elasticsearch允许我们存储文档相关索引的信息。可以通过使用内部_index字段做到这一点。试想一下，我们每天创建索引，对索引使用别名，且想知道返回文档存储在哪个索引中。在这种情况下，_index字段可以帮助确定文档源自哪个索引。

默认情况下，_index字段是禁用的。为了启用它，需要设置_index对象的enabled属性为true，如下所示：

```
{
  "book" : {
    "_index" : {
```

```
        "enabled" : true
      },
      "properties" : {
        .
        .
        .
      }
    }
  }
```

2.4.6 _size字段

默认情况下，_size字段未启用，这使我们能够自动索引_source字段的原始大小，并与文件一道存储。如果需要启用_size字段，则需添加_size属性并设置enabled属性值为true。此外，还可以通过使用通常的store属性设置_size字段使其被存储。因而，如果希望我们的映射包括_size字段并被存储，需要把映射文件修改成下面这样：

```
{
  "book" : {
    "_size" : {
      "enabled": true,
      "store" : "yes"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

2.4.7 _timestamp字段

默认情况下，禁用的_timestamp字段允许文档在被索引时存储。启用此功能很简单，只要添加_timestamp到映射文件并将enabled属性设置为true，如下所示：

```
{
  "book" : {
    "_timestamp" : {
      "enabled" : true
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

默认情况下，_timestamp字段未经分析编入索引，但不保存。你可以改变这两个参数以满足实际需求。除此之外，_timestamp字段与普通的日期字段一样，因而可以像处理寻常的、基于日期的字段一样改变它的格式。为此，只需要用所需的格式指定format属性（请参阅本章前面关于“日期”核心类型的描述，以了解更多关于日期格式的内容）。

另外，可以添加path属性，并将其设置为某字段的名称来获取日期，而不是在文件检索过程中自动创建_timestamp字段。因此，若希望_timestamp字段基于year字段，可修改映射文件，如下所示：


```
{
  "book" : {
    "_timestamp" : {
      "enabled" : true,
      "path" : "year",
      "format" : "YYYY"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```


你可能已经注意到，我们还修改了_timestamp字段的格式以匹配存储在year字段中的值。

> 如果你使用_timestamp字段，并让Elasticsearch自动创建它，则该字段的值会被设置为文档索引的时间。请注意，使用局部文档更新功能时，_timestamp字段也将被更新。

2.4.8 _ttl字段

_ttl字段表示time to live（生存时间），它允许定义文档的生命周期，周期结束之后文档会被自动删除。默认情况下，_ttl字段是禁用的。要启用，需要添加_ttl JSON对象并设置它的enabled属性为true，参考下面的例子：

```
{
  "book" : {
    "_ttl" : {
      "enabled" : true
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

如果要提供文件的默认过期时间，只需在_ttl字段定义中添加default属性和期望的过期时间。例如，要在30天后删除文件，将做以下设置：

```
{
  "book" : {
    "_ttl" : {
      "enabled" : true,
      "default" : "30d"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

默认情况下，该_ttl值未经分析即存储和索引。你可以改变这两个参数，但要记住这个字段要未经分析才能工作。

2.5　段合并介绍

1.1节提到段及其不变性，指出Lucene库以及Elasticsearch中一旦数据被写入某些结构，就不再改变。虽然这简化了一些东西，但是也引入了额外的工作，其中一个例子是删除。由于段是无法改变的，因而有关删除的相关信息必须单独存储并动态应用到搜索过程中。这样做是为了从返回结果中去除已删除的文件。另一个例子是文档无法修改（有些修改是可能的，例如修改数值型doc值）。当然，我们可以说，Elasticsearch支持文档更新（请参阅1.4节）。然而在底层，实际上是删除旧文档，再把更新内容的文档编入索引。

随着时间的推移和持续索引数据，越来越多的段被创建。因此，搜索性能可能会降低，而且索引可能比原先大，因为它仍含有被删除的文件。这使得段合并有了用武之地。

2.5.1　段合并

段合并的处理过程是：底层的Lucene库获取若干段，并在这些段信息的基础上创建一个新的段。由此产生的段拥有所有存储在原始段中的文档，除了被标记为删除的那些之外。合并操作之后，源段将从磁盘上删除。这是因为段合并在CPU和I/O的使用方面代价是相当高的，关键是要适当地控制这个过程被调用的时机和频率。

2.5.2　段合并的必要性

你可能会问为何要费心段合并。首先，构成索引的段越多，搜索速度越慢，需要使用的Lucene

内存也越多。其次，索引使用的磁盘空间和资源，例如文件描述符。如果从索引中删除许多文档，直到合并发生，则这些文档只是被标记为已删除，而没有在物理上删除。因而，大多数占用了CPU和内存的文档可能并不存在！好在Elasticsearch使用合理的默认值做段合并，这些默认值很可能不再需要做任何更改。

2.5.3 合并策略

合并策略描述了应执行合并过程的时机。Elasticsearch允许配置以下三种不同的策略。

- `tiered`：这是默认合并策略，合并尺寸大致相似的段，并考虑到每个层（tier）允许的最大段数量；
- `log_byte_size`：这个合并策略下，随着时间推移，将产生由索引大小的对数构成的索引，其中存在着一些较大的段以及一些合并因子较小的段等；
- `log_doc`：这个策略类似于`log_byte_size`合并策略，但根据索引中的文档数而非段的实际字节数来操作。

上述的每个策略都有自己的参数来定义行为以及可以覆盖的默认值。本书不做详细描述。如果你想了解更多，请查看*Mastering ElasticSearch*，或去http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-merge.html了解详情。

可使用`index.merge.policy.type`属性来设置想使用的合并策略，如下所示：

```
index.merge.policy.type: tiered
```

值得一提的是，索引创建后将无法再对值进行修改。

2.5.4 合并调度器

合并调度器指示Elasticsearch合并过程的方式，有如下两种可能。

- 并发合并调度器：这是默认的合并过程，在独立的线程中执行，定义好的线程数量可以并行合并。
- 串行合并调度器：这一合并过程在调用线程（即执行索引的线程）中执行。合并进程会一直阻塞线程直到合并完成。

调度器可使用`index.merge.scheduler.type`参数设置。若要使用串行合并调度器，需把参数值设为`serial`；若要使用并发调度器，则需把参数值设为`concurrent`。例如下面的调度程序：

```
index.merge.scheduler.type: concurrent
```

2.5.5 合并因子

每种合并策略都有好几种设置，我们已经说过在这里不一一介绍，但是合并因子是个例外，

它指定了索引过程中段合并的频率。合并因子较小时，搜索的速度更快，占用的内存也更少，但索引的速度会减慢；合并因子较大时，则索引速度加快，这是因为发生的合并较少，但搜索的速度变慢，占用的内存也会变大。对于log_byte_size和 log_doc合并策略，可以通过index.merge.policy.merge_factor参数来设置合并因子。

```
index.merge.policy.merge_factor: 10
```

上述例子将合并因子的值设置成10，10也是默认值。建议在批量索引时设置更高的merge_factor属性值，普通的索引维护则设置较低的属性值。

2.5.6　调节

之前提到过，合并可能需要很多的服务器资源。合并过程通常与其他操作并行执行，所以理论上不会产生太大的影响。在实践中，磁盘I/O操作的数量可能非常大，以致严重影响了整体性能。这时，调节（throttling）可以改善此情况。事实上，此功能既可用于限制合并的速度，也可以用于使用数据存储的所有操作。可以在Elasticsearch的配置文件中对调节进行设置（elasticsearch.yml文件），也可以动态使用设置API来设置（请参阅8.7.2节）。调整调节的设置有两个：type和value。

为了设置调节类型，设置indices.store.throttle.type属性值为下列值之一。

- none：该值定义不打开调节。
- merge：该值定义调节仅在合并过程中有效。
- all：该值定义调节在所有数据存储活动中有效。

第二个属性，indices.store.throttle.max_bytes_per_sec，描述了调节限制I/O操作的数量。顾名思义，该属性告诉我们每秒可以处理的字节数量。来看看以下配置：

```
indices.store.throttle.type: merge
indices.store.throttle.max_bytes_per_sec: 10mb
```

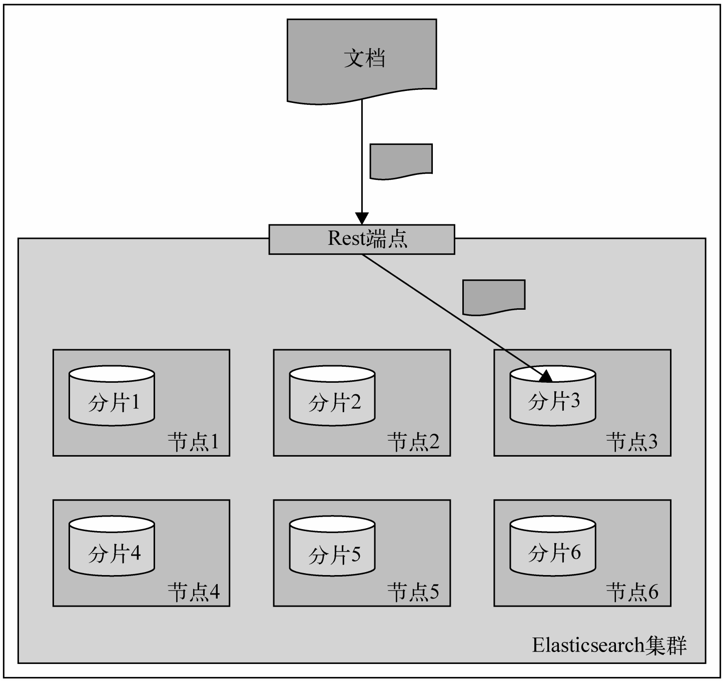
这个例子中，我们限制合并操作为每秒10 MB。默认情况下，Elasticsearch使用merge调节类型，max_bytes_per_sec属性设置值为20 mb。这意味着所有的合并操作都限于每秒20 MB。

2.6　路由介绍

默认情况下，Elasticsearch会在所有索引的分片中均匀地分配文档。然而，这并不总是理想情况。为了获得文档，Elasticsearch必须查询所有分片并合并结果。然而，如果你可以把数据按照一定的依据来划分（例如，客户端标识符），就可以使用一个强大的文档和查询分布控制机制：路由。简而言之，它允许选择用于索引和搜索数据的分片。

2.6.1　默认索引过程

在创建索引的过程中，当你发送文档时，Elasticsearch会根据文档的标识符，选择文档应编入索引的分片。默认情况下，Elasticsearch计算文档标识符的散列值，以此为基础将文档放置于一个可用的主分片上。接着，这些文档被重新分配至副本。下面的流程图简单演示了索引在默认情况下是如何工作的：



2.6.2　默认搜索过程

搜索与索引略有不同，在多数情况下，为了得到感兴趣的数据，你需要查询所有分片。试想一下，使用如下映射描述你的索引：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
        "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
        "index" : "analyzed" },
        "userId" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" }
```

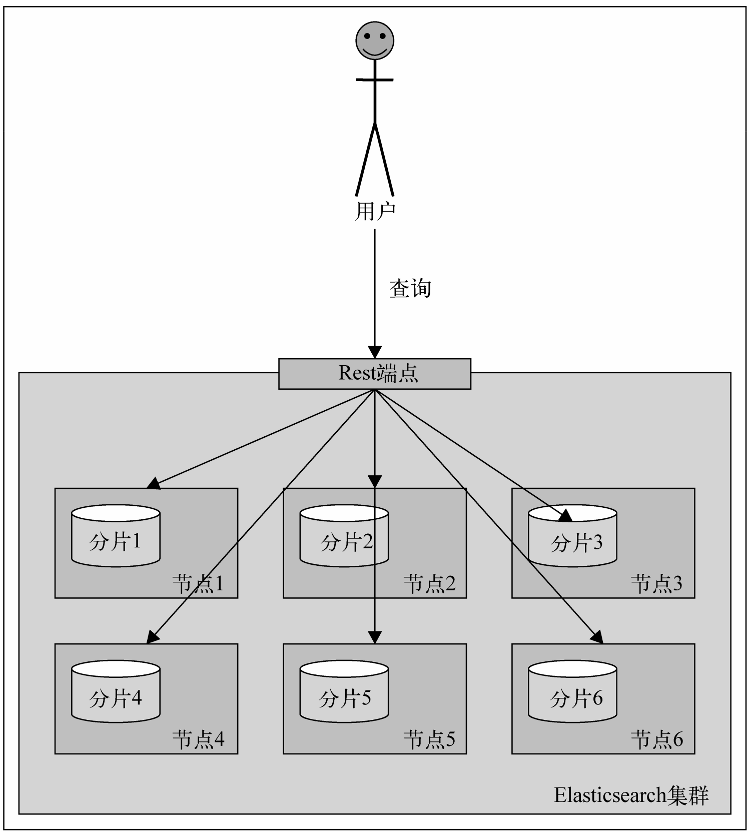
```
        }
      }
    }
  }
```

可以看到，索引包含了4个字段：标识符（id字段）、文档名称（name字段）、文档内容（contents字段）及文档所属用户的标识符（userId字段）。为了得到特定用户的所有文档（userId值为12），可以运行如下命令：

```
curl -XGET 'http://localhost:9200/posts/_search?q=userId:12'
```

一般而言，我们将查询发送到Elasticsearch的一个节点，Elasticsearch将会根据搜索类型（将在第3章讨论）来执行查询。这通常意味着它首先查询所有节点得到标识符和匹配文档的得分，接着发送一个内部查询，但仅发送到相关的分片（包含所需文档的分片），最后获取所需文档来构建响应。

以下视图简单演示了在搜索过程中默认路由的工作方式：



假使把单个用户的所有文档放置于单个分片之中，并对此分片查询，会出现什么情况？是否

对性能来说不明智？不，这种操作是相当便利的，也正是路由所允许的。

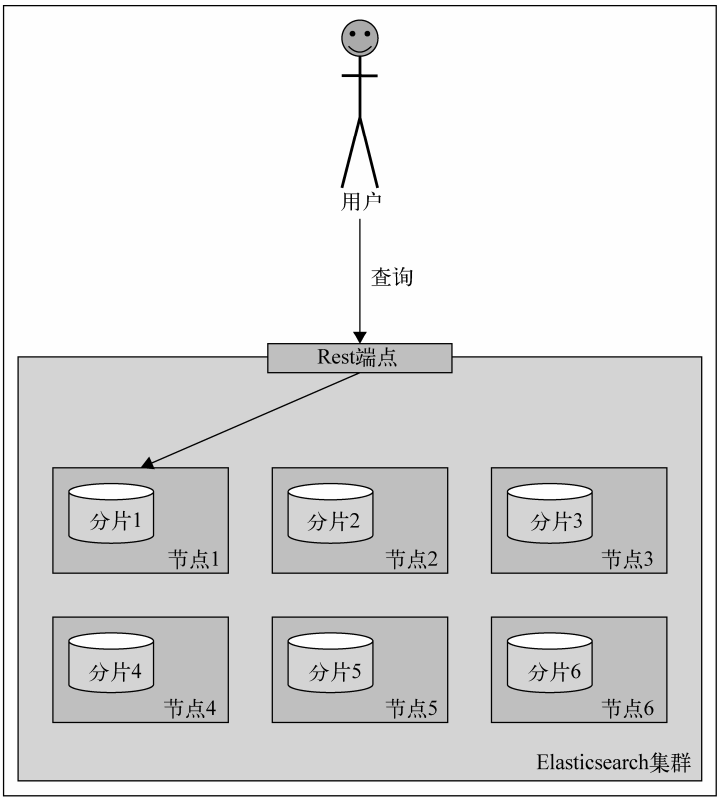
2.6.3 路由

路由可以控制文档和查询转发的目的分片。现在，你可能已经猜到了，可以在索引和查询时都指定路由值。事实上，如果你决定指定明确的路由值，可能会在索引和搜索过程中都这样做。

在我们的例子中，索引时使用`userId`值来设置路由，在搜索时也一样。你可以想象，对于相同的`userId`值，计算出的散列值是相同的，因而特定用户的所有文档将被放置在相同的分片中。在搜索中使用相同的属性值，则只需搜索单个分片而不是整个索引。

要记住，使用路由时，你仍然应该为与路由值相同的值添加一个过滤器。这是因为，路由值的数量或许会比索引分片的数量多。因此，一些不同的属性值可以指向相同的分片，如果你忽略过滤，得到的数据并非是路由的单个值，而是特定分片中驻留的所有路由值。

下图简单展示了给定路由值时，搜索是如何工作的：



如你所见，Elasticsearch将把查询发送到单个分片上。现在来看看如何指定路由值。

2.6.4 路由参数

最简单的方法（但并不总是最方便的一个）是使用路由参数来提供路由值。索引或查询时，你可以添加路由参数到HTTP，或使用你所选择的客户端库来设置。

所以，为了在前面所示的索引中建立一个示例文档，使用下列命令：

```
curl -XPUT 'http://localhost:9200/posts/post/1?routing=12' -d '{
  "id": "1",
  "name": "Test document",
  "contents": "Test document",
  "userId": "12"
}'
```

下面是使用路由参数的查询：

```
curl -XGET
  'http://localhost:9200/posts/_search?routing=12&q=userId:12'
```

可以看到，索引和查询时我们使用相同的路由值。这么做是因为我们知道在索引时设置的属性值为12，我们想要查询指向同一分片，因此需要使用完全相同的值。

请注意，你可以指定多个路由值，并由逗号分隔开来。如果还想在前面的查询中使用section参数（如果存在的话）来路由，并根据这个参数过滤，查询将会如下所示：

```
curl -XGET
  'http://localhost:9200/posts/_search?routing=12,
  6654&q=userId:12+AND+section:6654'
```



> 记住，路由值不是为了获取特定用户结果而唯一要指定的值。这是因为通常情况下分片很少有唯一的路由值。这意味着我们在单个分片中会有来自多个用户的数据。所以使用路由时，你还应该过滤结果。3.5节会介绍更多关于过滤的知识。

2.6.5 路由字段

为每个发送到Elasticsearch的请求指定路由值并不方便。事实上，在索引过程中，Elasticsearch允许指定一个字段，用该字段的值作为路由值。这样只需要在查询时提供路由参数。为此，在类型定义中需要添加以下代码：

```
"_routing" : {
  "required" : true,
  "path" : "userId"
}
```

上述定义意味着需要提供路由值（"required": true属性），否则，索引请求将失败。除此之外，我们还指定了path属性，说明文档的哪个字段值应被设置为路由值，在上述示例中，我们使用了userId字段值。这两个参数意味着用于索引的每个文档都需要定义userId字段。这

很便捷，因为现在使用批量索引时，无需单个分支的所有文档都使用相同的路由值（而设置路由参数的情况下，只能这样）。然而，请记住，在使用路由字段时，Elasticsearch需要一些额外的解析，因此比使用路由参数时慢一点。

添加路由部分后，整个更新的映射文件将如下所示：

```
{
  "mappings" : {
    "post" : {
      "_routing" : {
        "required" : true,
        "path" : "userId"
      },
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
        "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
        "index" : "analyzed" },
        "userId" : { "type" : "long", "store" : "yes",
        "precision_step" : "0" }
      }
    }
  }
}
```

如果想使用上述映射来创建posts索引，可使用下面的命令来为单个测试文档建立索引：

```
curl -XPOST 'localhost:9200/posts/post/1' -d '{
  "id":1,
  "name":"New post",
  "contents": "New test post",
  "userId":1234567
}'
```

Elasticsearch将使用1234567作为索引时的路由值。

2.7　小结

本章介绍了Elasticsearch索引的工作原理；如何创建自己的映射以定义索引结构，并使用它们创建索引；批量索引是什么以及如何使用，如何有效地索引数据；文档中可以存储哪些附加信息。除此之外，我们还了解了段合并是什么，如何配置，以及什么是调节。最后，学习了路由的使用和配置。

下一章的重点放在搜索上。我们将先介绍如何对Elasticsearch查询，有哪些基本的查询可用。之后，将使用过滤器，并了解它们为什么重要。接下来，我们将学习如何验证查询和使用高亮功能。最后，使用复合查询，探索查询的内部机制，对查询结果排序。

第 3 章

搜　　索

前一章介绍了Elasticsearch索引的工作原理，如何创建自定义映射，以及可以使用什么数据类型；还在索引中存储了附加的信息；使用了默认和非默认形式的路由。读完本章，我们将了解以下主题：

- 查询Elasticsearch并选择要返回的数据；
- Elasticsearch查询过程的工作机制；
- 了解Elasticsearch提供的基本查询；
- 筛选查询结果 ；
- 了解高亮的工作原理以及如何使用；
- 验证查询；
- 探索复合查询；
- 数据排序。

3.1　查询 Elasticsearch

到目前为止，我们使用了REST API和简单查询或GET请求来搜索数据。更改索引时，无论想执行的操作是更改映射还是文档索引化，都要用REST API向Elasticsearch发送JSON结构的数据。类似地，如果想发送的不是一个简单的查询，仍然把它封装为JSON结构并发送给Elasticsearch。这就是所谓的查询DSL。从更宏观的角度看，Elasticsearch支持两种类型的查询：基本查询和复合查询。基本查询，如词条查询用于查询实际数据，3.3节将介绍。第二种查询为复合查询，如布尔查询，可以合并多个查询，3.4节将讨论。

然而，这不是全部。除了这两种类型的查询，你还可以用过滤查询，根据一定的条件缩小查询结果。不像其他查询，筛选查询不会影响得分，而且通常非常高效。

更加复杂的情况，查询可以包含其他查询（别担心，我们将试着解释这个内容）。此外，一些查询可以包含过滤器，而其他查询可同时包含查询和过滤器。这并不是全部，但暂时先解释这些工作，3.4节和3.5节将详细介绍。

3.1.1　示例数据

如果没有特别说明，一些映射将用于本章的余下部分：

```
{
  "book" : {
    "_index" : {
      "enabled" : true
    },
    "_id" : {
      "index": "not_analyzed",
      "store" : "yes"
    },
    "properties" : {
      "author" : {
        "type" : "string"
      },
      "characters" : {
        "type" : "string"
      },
      "copies" : {
        "type" : "long",
        "ignore_malformed" : false
      },
      "otitle" : {
        "type" : "string"
      },
      "tags" : {
        "type" : "string"
      },
      "title" : {
        "type" : "string"
      },
      "year" : {
        "type" : "long",
        "ignore_malformed" : false,
        "index" : "analyzed"
      },
      "available" : {
        "type" : "boolean"
      }
    }
  }
}
```


> 如果没有特别说明，string类型的字段将被分析。

上述映射（保存为mapping.json文件）用来创建library索引。使用下面的命令来运行：

```
curl -XPOST 'localhost:9200/library'
curl -XPUT 'localhost:9200/library/book/_mapping' -d @mapping.json
```

如果没有特别说明，下面的数据在本章通用：

```
{ "index": {"_index": "library", "_type": "book", "_id": "1"}}
{ "title": "All Quiet on the Western Front","otitle": "Im Westen
  nichts Neues","author": "Erich Maria Remarque","year":
  1929,"characters": ["Paul Bäumer", "Albert Kropp", "Haie
  Westhus", "Fredrich Müller", "Stanislaus Katczinsky",
  "Tjaden"],"tags": ["novel"],"copies": 1, "available": true,
  "section" : 3}
{ "index": {"_index": "library", "_type": "book", "_id": "2"}}
{ "title": "Catch-22","author": "Joseph Heller","year":
  1961,"characters": ["John Yossarian", "Captain Aardvark",
  "Chaplain Tappman", "Colonel Cathcart", "Doctor
  Daneeka"],"tags": ["novel"],"copies": 6, "available" : false,
  "section" : 1}
{ "index": {"_index": "library", "_type": "book", "_id": "3"}}
{ "title": "The Complete Sherlock Holmes",
  "author": "Arthur Conan Doyle","year": 1936,"characters":
  ["Sherlock Holmes","Dr. Watson", "G. Lestrade"],"tags":
  [],"copies": 0, "available" : false, "section" : 12}
{ "index": {"_index": "library", "_type": "book", "_id": "4"}}
{ "title": "Crime and Punishment","otitle": "Преступлéние и
  наказáние","author": "Fyodor Dostoevsky","year":
  1886,"characters": ["Raskolnikov", "Sofia Semyonovna
  Marmeladova"],"tags": [],"copies": 0, "available" : true}
```

把上面数据保存在documents.json文件里，使用下面的命令来索引化：

```
curl -s -XPOST 'localhost:9200/_bulk' --data-binary @documents.json
```

该命令执行批量索引，你可以在2.3节中学习更多有关内容。

3.1.2　简单查询

查询Elasticsearch最简单的办法是使用URI请求查询，1.5节已经讨论过。例如，为了搜索`title`字段中的`crime`一词，使用下面的命令：

```
curl -XGET 'localhost:9200/library/book/_search?q=title:crime&pretty=true'
```

这种查询方式很简单，但比较局限。如果从Elasticsearch的查询DSL的视点来看，上面的查询是一种`query_string`查询，它查询`title`字段中含有`crime`一词的文档，可以这样写：

```
{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

采用查询DSL来发送查询有点不同，但也不是什么高深的东西。我们和以前一样发送HTTP GET请求到`_search`这个REST端点，并在请求主体中附上查询。来看看下面的命令：

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true' -d '{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}'
```

可以看到，我们使用请求体（-d参数）把整个JSON格式的查询发到Elasticsearch。pretty=true参数让Elasticsearch以更容易阅读的方式返回响应。上述命令的响应如下所示：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641, "_source" : { "title": "Crime and
        Punishment","otitle": "Преступлéние и наказáние","author":
        "Fyodor Dostoevsky","year": 1886,"characters":
        ["Raskolnikov", "Sofia Semyonovna Marmeladova"],"tags":
        [],"copies": 0, "available" : true}
    } ]
  }
}
```

很好！我们得到了使用查询DSL的第一个搜索结果。

3.1.3 分页和结果集大小

正如我们所期望的，Elasticsearch能控制想要的最多结果数以及想从哪个结果开始。下面是可以在请求体中添加的两个额外参数。

- from：该属性指定我们希望在结果中返回的起始文档。它的默认值是0，表示想要得到从第一个文档开始的结果。
- size：该属性指定了一次查询中返回的最大文档数，默认值为10。如果只对切面结果感兴趣，并不关心文档本身，可以把这个参数设置成0。

如果想让查询从第10个文档开始返回20个文档，可以发送如下查询：

```
{
  "from" : 9,
```

```
  "size" : 20,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

> 下载示例代码。如果你是用账号从http://www.packtpub.com买的书，可以到上面下载示例代码。如果你是从别的地方买的书，可以访问http://www.PacktPub.com/support并注册，把代码文件直接发到你的电子邮箱。

3.1.4　返回版本值

除了所有返回的信息以外，Elasticsearch还可以返回文档的版本。为此，需要在JSON对象的最上层添加`version`属性并把值设为`true`。所以，要求返回版本信息的查询，最终将如下所示:

```
{
  "version" : true,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

执行上面的查询后，得到如下结果:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
  "total" : 5,
  "successful" : 5,
  "failed" : 0
},
"hits" : {
  "total" : 1,
  "max_score" : 0.15342641,
  "hits" : [ {
    "_index" : "library",
    "_type" : "book",
    "_id" : "4",
    "_version" : 1,
    "_score" : 0.15342641, "_source" : { "title": "Crime and
      Punishment","otitle": "ПреступлéнХие и наказáние",
      "author": "Fyodor Dostoevsky","year": 1886,
      "characters": ["Raskolnikov", "Sofia Semyonovna
      Marmeladova"],"tags": [],"copies": 0, "available" : true}
   }]
  }
}
```

可以看到，`_version`属性出现在返回的唯一`hit`对象中。

3.1.5 限制得分

对于非标准用例，Elasticsearch提供一项功能，让我们可以根据文档需要满足的最低得分值，来过滤结果。为了用此功能，必须在JSON顶层提供min_score属性和最低得分值。例如，希望我们的查询只返回得分高于0.75的文档，发出以下查询：

```
{
  "min_score" : 0.75,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

执行后得到如下响应：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}
```

看下之前那个例子，文档的得分为0.153 426 41，比0.75低，所以这次没有得到任何文档。限制得分一般没太大意义，因为一般来说在查询之间比较得分很困难。也许在某些情况下，你将需要这个功能。

3.1.6 选择需要返回的字段

在请求主体中使用字段数组，可以定义在响应中包含哪些字段。记住，你只能返回那些在用于创建索引的映射中标记为存储的字段，或者你使用了_source字段（Elasticsearch使用_source字段提供存储字段）。因此，要让每个结果中的文档只返回title和year字段，发送下面的查询到Elasticsearch：

```
{
  "fields" : [ "title", "year" ],
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

在响应中，得到如下输出：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641,
      "fields" : {
        "title" : [ "Crime and Punishment" ],
        "year" : [ 1886 ]
      }
    } ]
  }
}
```

可以看到，一切按预期工作。与你分享以下3点：

- 如果没有定义fields数组，它将用默认值，如果有就返回_source字段；
- 如果使用_source字段，并且请求一个没有存储的字段，那么这个字段将从_source字段中提取（然而，这需要额外的处理）；
- 如果想返回所有的存储字段，只需传入星号（*）作为字段名字。

> 从性能的角度，返回_source字段比返回多个存储字段更好。

部分字段

除可以选择要返回哪些字段外，Elasticsearch允许使用所谓部分字段。可以通过它来控制字段是如何从_source字段加载的。Elasticsearch公开了部分字段对象的include和exclude属性，所以可以基于这些属性来包含或排除字段。例如，为了在查询中包括以titl开头且排除以chara开头的字段，发出以下查询：

```
{
  "partial_fields" : {
    "partial1" : {
      "include" : [ "titl*" ],
      "exclude" : [ "chara*" ]
    }
  },
  "query" : {
```

```
    "query_string" : { "query" : "title:crime" }
  }
}
```

3.1.7　使用脚本字段

可以在Elasticsearch中返回脚本计算字段：在JSON的查询对象中加上script_fields部分，添加上每个想返回的脚本值的名字。若要返回一个叫correctYear的值，它用year字段减去1800计算得来，运行以下查询：

```
{
  "script_fields" : {
    "correctYear" : {
      "script" : "doc['year'].value - 1800"
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

我们在上面的示例中使用了doc符号，它让我们捕获了返回结果，从而让脚本执行速度更快，但也导致了更高的内存消耗，并且限制了只能用单个字段的单个值。如果关心内存的使用，或者使用的是更复杂的字段值，可以用_source字段。使用此字段的查询如下所示：

```
{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - 1800"
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

这个查询将返回如下响应：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
        "_index" : "library",
```

```
          "_type" : "book",
          "_id" : "4",
          "_score" : 0.15342641,
          "fields" : {
          "correctYear" : [ 86 ]
        }
      } ]
    }
  }
```

可以看到，我们在响应中得到了correctYear这个计算字段。

传参数到脚本字段中

再看一个脚本字段的特性：可传入额外的参数。可以使用一个变量名称，并把值传入params节中，而不是直接把1800写在等式中。这样做以后，查询将如下所示：

```
{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - paramYear",
      "params" : {
      "paramYear" : 1800
      }
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

可以看到，我们在脚本的等式中添加了paramYear变量，并在params节中提供了变量的值。

5.2节将介绍关于脚本使用的更多内容。

3.2　理解查询过程

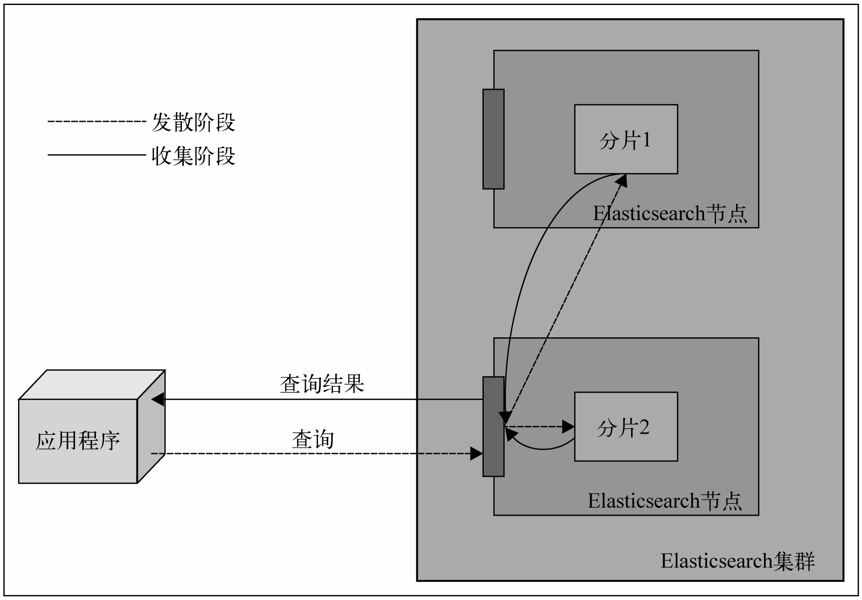
读完上一节后，我们知道了Elasticsearch的查询的工作原理。要知道在大多数情况下，Elasticsearch需要分散查询到多个节点中，得到结果，合并它们，再获取有关文档并返回结果。我们还没谈到的是另外三个定义查询行为的东西：查询重写、搜索类型和查询执行偏好。现在，将注意力集中在Elasticsearch的这些功能上，试着展示查询是如何工作的。

3.2.1　查询逻辑

Elasticsearch是一个分布式搜索引擎，因此提供的所有功能在性质上都必须是分布式的。查询也是如此。既然我们想讨论一些更高级的、关于如何控制查询过程的主题，首先需要知道它是

如何工作的。

默认情况下，如果我们什么都不改变，查询过程将分为两个阶段，如下图所示：



查询发送到Elasticsearch的其中一个节点，这时发生的是一个所谓的发散阶段。查询分布到建立过索引的所有分片上。如果它建立在5个分片和1个副本基础上，那么，这5个实体分片都会被查询（不需要同时查询分片及其副本，因为它们包含相同的数据）。每个查询的分片将只返回文档的标识符和得分。发送分散查询的节点将等待所有的分片完成它们的任务，收集结果并适当排序（在这种情况下，按得分从低到高）。

之后，将发送一个新的请求来生成搜索结果。然而，这次请求将只发送到那些持有响应所需文档的分片上。在大多数情况下，Elasticsearch不会把请求发送到所有的分片，而只是发送给其中的一部分。这是因为通常不需要整个查询结果，只要一部分。这一阶段被称为收集阶段（gather phase）。收集完所有文档，将建立最终响应，并返回查询结果。

当然，上述是Elasticsearch的默认行为，可以改变。以下部分将描述如何更改此行为。

3.2.2 搜索类型

Elasticsearch允许通过指定搜索类型来选择查询在内部如何处理。不同的搜索类型适合不同的情况；可以只在乎性能，但有时查询的关联性可能是最重要的因素。你应该记得每个分片是一个小的Lucene索引，为了返回更多相关的结果，频率等信息需要在分片之间传输。为了控制查询如何执行，可以使用`search_type`请求参数，并将其设置为下列值之一。

- query_then_fetch：第一步，执行查询得到对文档进行排序和分级所需信息。这一步在所有的分片上执行。然后，只在相关分片上查询文档的实际内容。不同于query_and_fetch，此查询类型返回结果的最大数量等于size参数的值。如果没有指定搜索类型，就默认使用这个类型，前面描述过。
- query_and_fetch：这通常是最快也最简单的搜索类型实现。查询在所有分片上并行执行（当然，任意一个主分片，只查询一个副本），所有分片返回等于size值的结果数。返回文档的最大数量等于size的值乘以分片的数量。
- dfs_query_and_fetch：这个跟query_and_fetch类似，但相比query_and_fetch，它包含一个额外阶段，在初始查询中执行分布式词频的计算，以得到返回文件的更精确的得分，从而让查询结果更相关。
- dfs_query_then_fetch：与前一个dfs_query_and_ fetch一样，dfs_query_then_fetch类似于相应的query_then_fetch，但比query_ then_fetch多了一个额外的阶段，就像dfs_query_and_fetch一样。
- count：这是一个特殊的搜索，只返回匹配查询的文档数。如果你只需要结果数量，而不关心文档，应该使用这个搜索类型。
- scan：这是另一个特殊的搜索类型，只有在要让查询返回大量结果时才用。它跟一般的查询有点不同，因为在发送第一个请求之后，Elasticsearch响应一个滚动标识符，类似于关系型数据库中的游标。所有查询需要在_search/scroll REST端点运行，并需要在请求主体中发送返回的滚动标识符。6.7节将介绍它的更多功能。

所以，如果想使用最简单的搜索类型，可以执行以下命令：

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true&search_
type=query_and_fetch' -d '{
  "query" : {
    "term" : { "title" : "crime" }
  }
}'
```

3.2.3　搜索执行偏好

除了可以控制查询是如何执行的，也可以控制在哪些分片上执行查询。默认情况下，Elasticsearch使用的分片和副本，既包含我们已经发送过请求的可用节点，又包括集群中的其他节点。而且，在大多数情况下，默认行为是最佳的查询首选方法。有时我们要更改默认行为，例如，可能希望只在主分片上执行搜索。为此，可以设置偏好请求参数，设为下面表中的其中一个值：

参 数 值	描　　述
_primary	只在主分片上执行搜索，不使用副本。当想使用索引中最近更新的、还没复制到副本中的信息，这个是很有用的
_primary_first	如果主分片可用，只在主分片上执行搜索，否则才在其他分片上执行

（续）

参数值	描　述
_local	在可能的情况下，只在发送请求的节点上的可用分片上执行搜索
_only_node:node_id	只在提供标识符的节点上执行搜索
_prefer_node:node_id	Elasticsearch将尝试在提供标识符的节点上执行搜素。如果该节点不可用，则使用其他的可用节点
_shards:1,2	Elasticsearch将在提供标识符的分片上执行操作(在这个例子中，分片1和2)。_shards参数可以和其他首选项合并，但_shards标识符必须在前面，比如_shards:1,2;_local
自定义值	可以传入任何自定义字符串值，具有相同值的请求将在相同的分片上执行

如果只想在本地分片上执行查询，可以执行如下的命令：

```
curl -XGET 'localhost:9200/library/_search?preference=_local' -d '{
  "query" : {
    "term" : { "title" : "crime" }
  }
}'
```

3.2.4 搜索分片 API

在讨论搜索偏好时，还想提到Elasticsearch所公开的搜索分片API。此API允许检查将执行查询的分片。需要在_search_shards REST端点执行这个API。若要查看查询如何执行，运行以下命令：

```
curl -XGET 'localhost:9200/library/_search_shards?pretty' -d
  '{"query":"match_all":{}}'
```

该命令将返回如下响应：

```
{
  "nodes" : {
    "N0iP_bH3QriX4NpqsqSUAg" : {
      "name" : "Oracle",
      "transport_address" : "inet[/192.168.1.19:9300]"
    }
  },
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
    "shard" : 0,
    "index" : "library"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
```

```
      "shard" : 1,
      "index" : "library"
    } ], [ {
      "state" : "STARTED",
      "primary" : true,
      "node" : "N0iP_bH3QriX4NpqsqSUAg",
      "relocating_node" : null,
      "shard" : 4,
      "index" : "library"
    } ], [ {
      "state" : "STARTED",
      "primary" : true,
      "node" : "N0iP_bH3QriX4NpqsqSUAg",
      "relocating_node" : null,
      "shard" : 3,
      "index" : "library"
    } ], [ {
      "state" : "STARTED",
      "primary" : true,
      "node" : "N0iP_bH3QriX4NpqsqSUAg",
      "relocating_node" : null,
      "shard" : 2,
      "index" : "library"
    } ] ]
  }
```

可以看到，在Elasticsearch返回的响应中，有关于在查询过程中使用的分片信息。当然，对于搜索分片API，可以使用所有参数，如`routing`或`preference`，看看它对搜索执行的影响。

3.3　基本查询

Elasticsearch具有广泛的搜索和数据分析能力，以不同的查询、筛选和聚合等形式公开。本节将集中讨论Elasticsearch提供的基本查询。

3.3.1　词条查询

词条查询是Elasticsearch中的一个简单查询。它仅匹配在给定字段中含有该词条的文档，而且是确切的、未经分析的词条。最简单的词条查询如下所示：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  }
}
```

上述查询将匹配`title`字段中含有crime一词的文档。记住，词条查询是未经分析的，因此需

要提供跟索引文档中的词条完全匹配的词条。请注意，在输入数据中，title字段含有Crime and Punishment，但我们使用小写开头的crime来搜索。因为Crime一词在建立索引时已经变成了crime。

除了想找的词条外，还可以在词条查询中包含加权属性，它影响给定词条的重要程度。5.1节将对此进行详细讨论。现在，只需记住它改变查询中给定词条的重要程度。

为了修改前面的查询，给它一个10.0的加权，可以发送如下查询：

```
{
  "query" : {
    "term" : {
      "title" : {
        "value" : "crime",
        "boost" : 10.0
      }
    }
  }
}
```

你可以看到，我们对查询做了点改变。不再是一个简单的词条，嵌套了一个新的JSON对象包含value属性和boost属性。value属性的值包含我们感兴趣的词条，boost属性的值是我们想使用的加权值。

3.3.2 多词条查询

多词条查询允许匹配那些在内容中含有某些词条的文档。词条查询允许匹配单个未经分析的词条，多词条查询可以用来匹配多个这样的词条。假设想得到所有在tags字段中含有novel或book的文档。运行以下查询来达到目的：

```
{
  "query" : {
    "terms" : {
      "tags" : [ "novel", "book" ],
      "minimum_match" : 1
    }
  }
}
```

上述查询返回在tags字段中包含一个或两个搜索词条的所有文档。为什么？这是因为我们把minimum_match属性设置为1；这意味着至少有1个词条应该匹配。如果想要查询匹配所有词条的文档，可以把minimum_match属性设置为2。

3.3.3 **match_all** 查询

match_all查询是Elasticsearch中最简单的查询之一。它使我们能够匹配索引中的所有文件。

如果想得到索引中的所有文档，只需运行以下查询：

```
{
  "query" : {
    "match_all" : {}
  }
}
```

也可以在查询中包含加权值，它将赋给所有跟它匹配的文档。比如，在match_all查询中给所有文档加上2.0的加权，可以发送以下查询：

```
{
  "query" : {
    "match_all" : {
      "boost" : 2.0
    }
   }
}
```

3.3.4　常用词查询

常用词查询是在没有使用停用词（stop word，http://en.wikipedia.org/wiki/Stop_words）的情况下，Elasticsearch为了提高常用词的查询相关性和精确性而提供的一个现代解决方案。例如，“crime and punishment”可以翻译成3个词查询，每一个都有性能上的成本（词越多，查询性能越低）。但“and”这个词非常常见，对文档得分的影响非常低。解决办法是常用词查询，将查询分为两组。第一组包含重要的词，出现的频率较低。第二组包含较高频率的、不那么重要的词。先执行第一个查询，Elasticsearch从第一组的所有词中计算分数。这样，通常都很重要的低频词总是被列入考虑范围。然后，Elasticsearch对第二组中的词执行二次查询，但只为与第一个查询中匹配的文档计算得分。这样只计算了相关文档的得分，实现了更高的性能。

一个常用词查询的例子如下：

```
{
  "query" : {
    "common" : {
      "title" : {
        "query" : "crime and punishment",
        "cutoff_frequency" : 0.001
      }
    }
  }
}
```

查询可使用下列参数。

- query：这个参数定义了实际的查询内容。
- cutoff_frequency：这个参数定义一个百分比（0.001表示0.1%）或一个绝对值（当此

属性值>=1时）。这个值用来构建高、低频词组。此参数设置为0.001意味着频率<=0.1%的词将出现在低频词组中。

- low_freq_operator：这个参数可以设为or或and，默认是or。它用来指定为低频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配，应该把它设置为and。
- high_freq_operator：这个参数可以设为or或and，默认是o。它用来指定为高频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配，那么应该把它设置为and。
- minimum_should_match：不使用low_freq_operator和high_freq_operator参数的话，可以使用minimum_should_match参数。和其他查询一样，它允许指定匹配的文档中应该出现的查询词的最小个数。
- boost：这个参数定义了赋给文档得分的加权值。
- analyzer：这个参数定义了分析查询文本时用到的分析器名称。默认值为default analyzer。
- disable_coord：此参数的值默认为false，它允许启用或禁用分数因子的计算，该计算基于文档中包含的所有查询词的分数。把它设置为true，得分不那么精确，但查询将稍快。

> 不像词条查询和多词条查询，常用词查询是经过Elasticsearch分析的。

3.3.5 match 查询

match查询把query参数中的值拿出来，加以分析，然后构建相应的查询。使用match查询时，Elasticsearch将对一个字段选择合适的分析器，所以可以确定，传给match查询的词条将被建立索引时相同的分析器处理。请记住，match查询（以及将在稍后解释的multi_match查询）不支持Lucene查询语法。但是，它是完全符合搜索需求的一个查询处理器。最简单也是默认的match查询如下所示：

```
{
  "query" : {
    "match" : {
      "title" : "crime and punishment"
    }
  }
}
```

上面的查询将匹配所有在title字段含有crime、and或punishment词条的文档。这只是最简单的一个查询，现在来讨论match查询的几种类型。

1. 布尔值匹配查询

布尔匹配查询分析提供的文本，然后做出布尔查询。有几个参数允许控制布尔查询匹配行为，如下所示。

- operator：此参数可以接受or和and，控制用来连接创建的布尔条件的布尔运算符。默认值是or。如果希望查询中的所有条件都匹配，可以使用and运算符。
- analyzer：这个参数定义了分析查询文本时用到的分析器的名字。默认值为default analyzer。
- fuzziness：可以通过提供此参数的值来构建模糊查询（fuzzy query）。它为字符串类型提供从0.0到1.0的值。构造模糊查询时，该参数将用来设置相似性。
- prefix_length：此参数可以控制模糊查询的行为。有关此参数值的更多信息，请参阅3.3.11节。
- max_expansions：此参数可以控制模糊查询的行为。有关此参数值的更多信息，请参阅3.3.11节。
- zero_terms_query：该参数允许指定当所有的词条都被分析器移除时（例如，因为停止词），查询的行为。它可以被设置为none或all，默认值是none。在分析器移除所有查询词条时，该参数设置为none，将没有文档返回；设置为all，则将返回所有文档。
- cutoff_frequency：该参数允许将查询分解成两组：一组低频词和一组高频词。参阅3.3.4节，看看这个参数怎么用。

这些参数应该封装在运行查询的字段名称里。所以如果想对title字段运行一个简单的布尔匹配查询，发送如下查询：

```
{
  "query" : {
    "match" : {
      "title" : {
      "query" : "crime and punishment",
      "operator" : "and"
      }
    }
  }
}
```

2. match_phrase查询

match_phrase查询类似于布尔查询，不同的是，它从分析后的文本中构建短语查询，而不是布尔子句。该查询可以使用下面几种参数。

- slop：这是一个整数值，该值定义了文本查询中的词条和词条之间可以有多少个未知词条，以被视为跟一个短语匹配。此参数的默认值是0，这意味着，不允许有额外的词条[1]。

---

① slop为1时，“a b”和“a and b”被视为匹配。——译者注

- analyzer：这个参数定义了分析查询文本时用到的分析器的名字。默认值为default analyzer。

下面是一段对title字段进行match_phrase查询的示例代码：

```
{
  "query" : {
    "match_phrase" : {
      "title" : {
        "query" : "crime punishment",
        "slop" : 1
      }
    }
   }
}
```

注意，我们从查询中移除了and一词，但因为slop参数设置为1，它仍将匹配我们的文档。

3. **match_phrase_prefix查询**

match_query查询的最后一种类型是match_phrase_prefix查询。此查询跟match_phrase查询几乎一样，但除此之外，它允许查询文本的最后一个词条只做前缀匹配。此外，除了match_phrase查询公开的参数，还公开了一个额外参数max_expansions。这个参数控制有多少前缀将被重写成最后的词条。我们的示例查询若改用match_phrase前缀来写，将如下所示：

```
{
    "query" : {
      "match_phrase_prefix" : {
        "title" : {
        "query" : "crime and punishm",
        "slop" : 1,
        "max_expansions" : 20
      }
    }
  }
}
```

注意，我们没有提供完整的“crime and punishment”短语，而只是提供“crime and punishm”，该查询仍将匹配我们的文档。

3.3.6 multi_match 查询

multi_match查询和match查询一样，不同的是它不是针对单个字段，而是可以通过fields参数针对多个字段查询。当然，match查询中可以使用的所有参数同样可以在multi_match查询中使用。所以，如果想修改match查询，让它针对title和otitle字段运行，那么运行以下查询：

```
{
  "query" : {
    "multi_match" : {
```

```
        "query" : "crime punishment",
        "fields" : [ "title", "otitle" ]
      }
    }
}
```

除了之前提到的参数，multi_match查询还可以使用以下额外的参数来控制它的行为。

- use_dis_max：该参数定义一个布尔值，设置为true时，使用析取最大分查询，设置为false时，使用布尔查询。默认值为true。3.3.18节将讨论更多细节。
- tie_breaker：只有在use_dis_max参数设为true时才会使用这个参数。它指定低分数项和最高分数项之间的平衡。3.3.18节将介绍更多细节。

3.3.7 **query_string** 查询

相比其他可用的查询，query_string查询支持全部的Apache Lucene查询语法，1.5.3节讨论过。它使用一个查询解析器把提供的文本构建成实际的查询，例子如下所示：

```
{
  "query" : {
    "query_string" : {
     "query" : "title:crime^10 +title:punishment -otitle:cat
      +author:(+Fyodor +dostoevsky)",
      "default_field" : "title"
    }
  }
}
```

我们已经熟悉了Lucene查询语法的基础知识，所以可以讨论上述查询的工作原理。正如你所看到的，我们想得到在title字段中包含crime词条的文档，并且这些文档应该有10的加权。接下来，我们希望文档在title字段中包含punishment，而在otitle字段中不包含cat。最后，告诉Lucene我们只希望文档的author字段中包含Fyodor和dostoevsky词条。

像大多数Elasticsearch查询一样，query_string提供下列参数控制查询行为。

- query：此参数指定查询文本。
- default_field：此参数指定默认的查询字段，默认值由index.query.default_field属性指定，默认为_all。
- default_operator：此参数指定默认的逻辑运算符（or或and），默认值是or。
- allow_leading_wildcard：此参数指定是否允许通配符作为词条的第一个字符，默认值为true。
- lowercase_expand_terms：此参数指定查询重写是否把词条变成小写，默认值为true，意味着重写后的词条将小写。
- enable_position_increments：此参数指定查询结果中的位置增量是否打开，默认值是true。

- fuzzy_max_expansions：使用模糊查询时，此参数指定模糊查询可被扩展到的最大词条数，默认值是50。
- fuzzy_prefix_length：此参数指定生成的模糊查询中的前缀长度，默认值为0。欲了解更多信息，请参阅3.3.11节。
- fuzzy_min_sim：此参数指定模糊查询的最小相似度，默认值为0.5。欲了解更多信息，请参阅3.3.11节。
- phrase_slop：此参数指定短语溢出值，默认值为0。欲了解更多信息，请参阅3.3.5节。
- boost：此参数指定使用的加权值，默认值为1.0。
- analyze_wildcard：此参数指定是否应该分析通配符查询生成的词条，默认为false，意味着词条不会被分析。
- auto_generate_phrase_queries：此参数指定是否自动生成短语查询。其默认值为false，这意味着不会自动生成。
- minimum_should_match：此参数控制有多少生成的Boolean should子句必须与文档匹配，才能认为它是匹配的。它可以是百分比，例如50%，这意味着至少有50%的给定词条必须匹配。它也可以是整数值，如2，这意味着至少2个词条必须匹配。
- lenient：此参数的取值true或false。如果设置为true，格式方面的失败将被忽略。

3

DisMax是Disjunction Max的缩写。Disjunction指搜索执行可以跨多个字段，每个字段可以给予不同的权重。Max意味着，对于给定词条，只有最高分会包括在最后的文档评分中，而不是所有包含该词条的所有字段分数之和（简单的布尔查询才会这样）。

注意，Elasticsearch可以重写query_string查询，正因为如此，Elasticsearch使我们能够传递额外的参数来控制重写方法。有关此过程的详细信息，请参阅3.2节。

针对多字段的query_string查询

针对多个字段做query_string查询是可能的。为此，需要在查询主体中提供一个fields参数，它是个持有字段名称的数组。有两种方法针对多个字段运行query_string查询；默认方法是采用布尔查询来构造查询，另一种是使用最大分查询。

使用最大分查询要在查询主体中添加use_dis_max属性并将其设置为true。示例查询如下：

```
{
  "query" : {
    "query_string" : {
      "query" : "crime punishment",
      "fields" : [ "title", "otitle" ],
      "use_dis_max" : true
    }
  }
}
```

3.3.8 simple_query_string 查询

simple_query_string查询使用Lucene的最新查询解析器之一：SimpleQueryParser。类似字符串查询，它接受Lucene查询语法；然而不同的是，simple_query_string查询在解析错误时不会抛出异常。它丢弃查询无效的部分，执行其余部分，示例如下：

```
{
  "query" : {
    "simple_query_string" : {
      "query" : "title:crime^10 +title:punishment -otitle:cat
        +author:(+Fyodor +dostoevsky)",
        "default_operator" : "and"
    }
  }
}
```

3.3.9 标识符查询

标识符查询是一个简单的查询，仅用提供的标识符来过滤返回的文档。此查询针对内部的_uid字段运行，所以它不需要启用_id字段。最简单的版本类似于下面的代码：

```
{
  "query" : {
    "ids" : {
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

此查询只返回具有values数组中一个标识符的文档。也可以把标识符查询变得复杂一点，限制文档为特定的类型。例如，只包括book类型的文档，发出以下查询：

```
{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

可以看到，我们在查询中添加了type属性，并设置其值为我们感兴趣的类型。

3.3.10 前缀查询

前缀查询在配置方面来说跟词条查询类似。前缀查询能让我们匹配这样的文档：它们的特定字段以给定的前缀开始。例如，想找到所有title字段以cri开始的文档，可以运行以下查询：

```
{
  "query" : {
    "prefix" : {
      "title" : "cri"
    }
  }
}
```

与词条查询类似，还可以在前缀查询中包含加权属性；这将影响到给定前缀的重要性。例如，改变之前的查询，并给它增加3.0的加权，发出以下查询：

```
{
  "query" : {
    "prefix" : {
      "title" : {
        "value" : "cri",
        "boost" : 3.0
      }
    }
  }
}
```


> Elasticsearch会把前缀查询重写，也允许我们传递额外的参数来控制重写方法。有关此过程的详细信息，请参阅3.2节。

3.3.11 **fuzzy_like_this** 查询

fuzzy_like_this查询类似于more_like_this查询。它查找所有与提供的文本类似的文档，但是它有点不同于more_like_this查询。它利用模糊字符串并选择生成的最佳差分词条。如果针对title和otitle字段的fuzzy_like_this查询来查找所有类似于crime punishment的文档，可以运行以下查询：

```
{
  "query" : {
    "fuzzy_like_this" : {
      "fields" : ["title", "otitle"],
      "like_text" : "crime punishment"
    }
  }
}
```

fuzzy_like_this查询支持以下查询参数。

- fields：此参数定义应该执行查询的字段数组，默认值是_all字段。
- like_text：这是一个必需参数，包含用来跟文档比较的文本。
- ignore_tf：此参数指定在相似度计算期间，是否应忽略词频，默认值为false，意味着将使用词频。

- ❑ max_query_terms：此参数指定生成的查询中能包括的最大查询词条数，默认值为25。
- ❑ min_similarity：此参数指定差分词条（differencing terms）应该有的最小相似性，默认值为0.5。
- ❑ prefix_length：此参数指定差分词条的公共前缀长度，默认值为0。
- ❑ boost：此参数指定使用的加权值，默认值为1.0。
- ❑ analyzer：这个参数定义了分析所提供文本时用到的分析器名称。

3.3.12　fuzzy_like_this_field 查询

fuzzy_like_this_field查询和fuzzy_like_this查询类似，但它只能对应单个字段。正因为如此，它不支持多字段属性。作为替代，应该把查询参数封装到字段名称中。查询title字段的一个示例查询类似于下面这样：

```
{
  "query" : {
    "fuzzy_like_this_field" : {
      "title" : {
        "like_text" : "crime and punishment"
      }
    }
  }
}
```

fuzzy_like_this查询的其他所有参数也可以用在fuzzy_like_this_field中。

3.3.13　fuzzy 查询

fuzzy查询是模糊查询中的第三种类型，它基于编辑距离算法来匹配文档。编辑距离的计算基于我们提供的查询词条和被搜索文档。此查询很占用CPU资源，但当需要模糊匹配时它很有用，例如，当用户拼写错误时。在我们的示例中，假设用户向搜索框中输入单词crme，而不是crime，运行模糊查询的最简单形式如下所示：

```
{
  "query" : {
    "fuzzy" : {
      "title" : "crme"
    }
  }
}
```

查询响应如下所示：

```
{
  "took" : 1,
  "timed_out" : false,
```

```
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641, "_source" : { "title": "Crime and
        Punishment","otitle": "Преступлéние и наказáние",
        "author": "Fyodor Dostoevsky","year": 1886,
        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"],"tags": [],"copies": 0, "available" : true}
    } ]
  }
}
```

即使犯了一个拼写错误，Elasticsearch仍然设法找到我们感兴趣的文档。

可以使用下面的参数来控制fuzzy查询的行为。

- value：此参数指定了实际的查询。
- boost：此参数指定了查询的加权值，默认为1.0。
- min_similarity：此参数指定了一个词条被算作匹配所必须拥有的最小相似度。对字符串字段来说，这个值应该在0到1之间，包含0和1。对于数值型字段，这个值可以大于1，比如查询值是20，min_similarity设为3，则可以得到17~23的值。对于日期字段，可以把min_similarity参数值设为1d、2d、1m等，分别表示1天、2天、1个月。
- prefix_length：此参数指定差分词条的公共前缀长度，默认值为0。
- max_expansions：此参数指定查询可被扩展到的最大词条数，默认值是无限制。

参数应该封装在查询针对的字段名称里。所以如果想修改前面的查询，并添加额外的参数，查询将如下所示：

```
{
  "query" : {
    "fuzzy" : {
      "title" : {
        "value" : "crme",
        "min_similarity" : 0.2
      }
    }
  }
}
```

3.3.14　通配符查询

通配符查询允许我们在查询值中使用*和?等通配符。此外，通配符查询跟词条查询在内容方面非常类似。可以发送一下查询，来匹配所有包含cr?me词条的文档，这里?表示任意字符：

```
{
  "query" : {
    "wildcard" : {
      "title" : "cr?me"
    }
  }
}
```

这将匹配title字段中包含与cr?me匹配的词条的所有文档。然后，你还可以在通配符查询中包含加权属性；它将影响每个与给定值匹配的词条的重要性。如果要改变之前的查询，给它一个20.0的加权，可以发出以下查询：

```
{
  "query" : {
    "wildcard" : {
      "title" : {
        "value" : "cr?me",
        "boost" : 20.0
      }
    }
  }
}
```

> 请注意，通配符查询不太注重性能，在可能时应尽量避免，特别是要避免前缀通配符（以通配符开始的词条）。此外，请注意Elasticsearch会重写通配符查询，因此Elasticsearch允许通过一个额外的参数控制重写方法。有关此过程的详细信息，请参阅3.2节。

3.3.15　**more_like_this** 查询

more_like_this查询让我们能够得到与提供的文本类似的文档。Elasticsearch支持几个参数来定义more_like_this查询如何工作，如下所示。

- fields：此参数定义应该执行查询的字段数组，默认值是_all字段。
- like_text：这是一个必需的参数，包含用来跟文档比较的文本。
- percent_terms_to_match：此参数定义了文档需要有多少百分比的词条与查询匹配才能认为是类似的，默认值为0.3，意思是30%。
- min_term_freq：此参数定义了文档中词条的最低词频，低于此频率的词条将被忽略，默认值为2。

- max_query_terms：此参数指定生成的查询中能包括的最大查询词条数，默认值为25。值越大，精度越大，但性能也越低。
- stop_words：此参数定义了一个单词的数组，当比较文档和查询时，这些单词将被忽略，默认值为空数组。
- min_doc_freq：此参数定义了包含某词条的文档的最小数目，低于此数目时，该词条将被忽略，默认值为5，意味着一个词条至少应该出现在5个文档中，才不会被忽略。
- max_doc_freq：此参数定义了包含某词条的文档的最大数目，高于此数目时，该词条将被忽略，默认值为无限制。
- min_word_len：此参数定义了单词的最小长度，低于此长度的单词将被忽略，默认值为0。
- max_word_len：此参数定义了单词的最大长度，高于此长度的单词将被忽略，默认值为无限制。
- boost_terms：此参数定义了用于每个词条的加权值，默认值为1。
- boost：此参数定义了用于查询的加权值，默认值为1。
- analyzer：此参数指定了针对我们提供的文本的分析器名称。

more_like_this查询的一个示例如下所示：

```
{
  "query" : {
    "more_like_this" : {
      "fields" : [ "title", "otitle" ],
      "like_text" : "crime and punishment",
      "min_term_freq" : 1,
      "min_doc_freq" : 1
    }
  }
}
```

3.3.16 more_like_this_filed 查询

more_like_this_field查询与more_like_this查询类似，但它只能针对单个字段。正因为如此，它不支持多字段属性。我们把查询参数封装到要查询的字段名字中，而不是指定fields参数。所以，一个查询title字段的示例查询如下：

```
{
  "query" : {
    "more_like_this_field" : {
      "title" : {
        "like_text" : "crime and punishment",
        "min_term_freq" : 1,
        "min_doc_freq" : 1
      }
    }
  }
}
```

more_like_this查询中的所有其他参数都可以同样的方式作用于more_like_this_filed查询。

3.3.17　范围查询

范围查询使我们能够找到在某一字段值在某个范围里的文档，字段可以是数值型，也可以是基于字符串的（将映射到一个不同的Apache Lucene查询）。范围查询只能针对单个字段，查询参数应封装在字段名称中。范围查询支持以下参数。

- gte：范围查询将匹配字段值大于或等于此参数值的文档。
- gt：范围查询将匹配字段值大于此参数值的文档。
- lte：范围查询将匹配字段值小于或等于此参数值的文档。
- lt：范围查询将匹配字段值小于此参数值的文档。

因此，举例来说，要找到year字段从1700到1900的所有图书，可以运行以下查询：

```
{
  "query" : {
    "range" : {
      "year" : {
        "gte" : 1700,
        "lte" : 1900
      }
    }
  }
}
```

3.3.18　最大分查询

最大分查询非常有用，因为它会生成一个由所有子查询返回的文档组成的并集并将它返回。这个查询好的一面是，我们可以控制较低得分的子查询对文档最后得分的影响。

文档的最后得分是这样计算的：最高分数的子查询的得分之和，加上其余子查询的得分之和乘以tie参数的值。所以，可以通过tie_breaker参数来控制较低得分的查询对最后得分的影响。把tie_breaker设为1.0，得到确切的总和；把tie_breaker设为0.1，结果，除最高得分的查询外，只有所有查询总得分的10%被加到最后得分里。

一个最大分查询的示例如下所示：

```
{
  "query" : {
    "dismax" : {
      "tie_breaker" : 0.99,
      "boost" : 10.0,
      "queries" : [
        {
```

```
            "match" : {
              "title" : "crime"
            }
          },
          {
            "match" : {
              "author" : "fyodor"
            }
          }
        ]
      }
    }
  }
```

可以看到，我们在查询中包含了tie_breaker和boost参数。此外，在queries参数中指定了一组查询，这些查询将执行并产生结果文档的并集。

3.3.19　正则表达式查询

通过正则表达式查询，可以使用正则表达式来查询文本。请记住，此类查询的性能取决于所选的正则表达式。如果我们的正则表达式匹配许多词条，查询将很慢。一般规则是，正则表达式匹配的词条数越高，查询越慢。

正则表达式查询示例如下所示：

```
{
  "query" : {
    "regexp" : {
      "title" : {
        "value" : "cr.m[ae]",
        "boost" : 10.0
      }
    }
  }
}
```

上述查询将被Elasticsearch重写成若干个词条查询，根据索引中匹配给定正则表达式的内容。查询中加权参数指定了生成的查询将使用的加权值。

> 完整的Elasticsearch支持的正则表达式语法可以在这里找到：http://www.lasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-regexp-uery.html#egexp-syntax。

3.4　复合查询

3.3节讨论了Elasticsearch公开的最简单查询。然而，Elasticsearch提供的不只如此。顾名思义，

复合查询就是支持可以把多个查询连接起来，或者改变其他查询的行为。你可能好奇自己是否需要这样的功能。用一个简单的练习来确定：结合一个简单的词条查询和一个短语查询，得到更好的搜索结果。

3.4.1　布尔查询

可以通过布尔查询来封装无限数量的查询，并通过下面描述的节点之一使用一个逻辑值来连接它们。

- should：被它封装的布尔查询可能被匹配，也可能不被匹配。被匹配的should节点数目由minimum_should_match参数控制。
- must：被它封装的布尔查询必须被匹配，文档才会返回。
- must_not：被它封装的布尔查询必须不被匹配，文档才会返回。

上述每个节点都可以在单个布尔查询中出现多次。这允许建立非常复杂的查询，有多个嵌套级别（在一个布尔查询中包含另一个布尔查询）。记住，结果文档的得分将由文档匹配的所有封装的查询得分总和计算得到。

除了上述部分以外，还可以在查询主体中添加以下参数控制其行为。

- boost：此参数指定了查询使用的加权值，默认为1.0。加权值越高，匹配文档的得分越高。
- minimum_should_match：此参数的值描述了文档被视为匹配时，应该匹配的should子句的最少数量。举例来说，它可以是个整数值，比如2，也可以是个百分比，比如75%。更多有关信息，参见http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/uery-dsl-minimum-should-match.html。
- disable_coord：此参数的默认值为false，允许启用或禁用分数因子的计算，该计算是基于文档包含的所有查询词条。如果得分不必太精确，但要查询快点，那么应该将它设置为true。

假设我们想要找到所有这样的文档：在title字段中含有crime词条，并且year字段可以在也可以不在1900~2000的范围里，在otitle字段中不可以包含nothing词条。用布尔查询的话，类似于下面的代码：

```
{
  "query" : {
    "bool" : {
      "must" : {
      "term" : {
        "title" : "crime"
      }
    },
    "should" : {
```

```
          "range" : {
            "year" : {
              "from" : 1900,
             "to" : 2000
            }
          }
        },
        "must_not" : {
          "term" : {
            "otitle" : "nothing"
          }
        }
      }
    }
  }
```


> 注意：must，should和must_not节点可以包含单一查询，也可以包含多个查询。

3.4.2 加权查询

加权查询封装了两个查询，并且降低其中一个查询返回文档的得分。加权查询中有三个节点需要定义：positive部分，包含所返回文档得分不会被改变的查询；negative部分，返回的文档得分将被降低；negative_boost部分，包含用来降低negative部分查询得分的加权值。

加权查询的优点是，positive部分和negative部分包含的查询结果都会出现在搜索结果中，而某些查询的得分将被降低。如果使用布尔查询的must_not节点，将得不到这样的结果。

假设我们想要一个简单的词条查询，查询title字段中含有crime词条，希望这样的文档得分不被改变，同时要year字段在1800~1900内的文档，但这样文档的得分要有一个0.5的加权。这样的查询如下所示：

```
{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "title" : "crime"
        }
      },
      "negative" : {
        "range" : {
          "year" : {
            "from" : 1800,
            "to" : 1900
          }
        }
      },
```

```
            "negative_boost" : 0.5
        }
    }
}
```

3.4.3 constant_score 查询

constant_score查询封装了另一个查询（或过滤），并为每一个所封装查询（或过滤）返回的文档返回一个常量得分。它允许我们严格控制与一个查询或过滤匹配的文档得分。如果希望title字段包含crime词条的所有文档的得分为2.0，可以发出以下查询：

```
{
  "query" : {
    "constant_score" : {
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "boost" : 2.0
    }
  }
}
```

3.4.4 索引查询

当针对多个索引执行查询时，索引查询很有用。可以通过indices属性提供一个索引的数组以及两个查询，一个通过query属性指定，将执行在指定的索引列表上；另一个通过no_match_query属性指定，将执行在其他所有索引上。假设我们有一个别名：books，它持有两个索引：library和users，我们希望使用别名；然而，我们希望在那些索引上执行不同的查询，为此，发送以下查询：

```
{
  "query" : {
    "indices" : {
      "indices" : [ "library" ],
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "no_match_query" : {
        "term" : {
          "user" : "crime"
        }
      }
    }
  }
}
```

上述查询中，query属性中的查询将执行在library索引上，no_match_query属性中的查询将执行在集群中其他所有索引上。

no_match_query属性也可以是个字符串值，而不是一个查询。这个字符串值可以是all或者none，默认是all。设置为all，索引中不匹配的所有文档都会返回；设置为none，索引中不匹配的文档将不会返回。

> Elasticsearch公开的一些查询，如custom_score查询、custom_boost_factor查询和custom_filters_scores查询，已经被function_score查询取代，5.4.3节将描述。我们决定省略这些查询的描述，因为它们在Elasticsearch的未来版本中可能会被删除。

3.5　查询结果的过滤

本书已经介绍了如何使用不同的条件和查询来构建查询并搜索数据。我们还熟知了评分（参见1.1.3节），它告诉我们在给定的查询中，哪些文档更重要以及查询文本如何影响排序。然而，有时我们可能要在不影响最后分数的情况下，选择索引中的某个子集，这就要使用过滤器（当然不是唯一的原因）。

老实说，应该尽可能使用过滤器。过滤器不影响评分，而得分计算让搜索变得复杂，而且需要CPU资源。另一方面，过滤是一种相对简单的操作。由于过滤应用在整个索引的内容上，过滤的结果独立于找到的文档，也独立于文档之间的关系。过滤器很容易被缓存，从而进一步提高过滤查询的整体性能。

以下有关过滤器的章节中，我们使用post_filter参数保持例子尽可能地简单。然而，请记住，如果可能，应该总是使用filtered查询，而不是post_filter，因为filtered执行起来更快。

3.5.1　使用过滤器

在任何搜索中使用过滤器，只需在于query节点相同级别上添加一个filter节点。如果你只想要过滤器，也可以完全省略query节点。来看一个示例查询，在title字段搜索Catch-22并向其添加过滤器，如下所示：

```
{
  "query" : {
    "match" : { "title" : "Catch-22" }
  },
  "post_filter" : {
    "term" : { "year" : 1961 }
  }
}
```

它返回给定title的所有文档，但结果缩小到仅在1961年出版的书。还有一种在查询中包含过滤器的方法：使用filtered查询。所以前面的查询可以重写如下：

```
{
  "query": {
    "filtered" : {
      "query" : {
        "match" : { "title" : "Catch-22" }
      },
      "filter" : {
        "term" : { "year" : 1961 }
      }
    }
  }
}
```

如果发送curl -XGET localhost:9200/library/book/_search?pretty -d @query.son命令来执行两个查询，你将看到它们的响应完全一样（可能除了响应时间）：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.2712221,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 0.2712221, "_source" : { "title": "Catch-
        22","author": "Joseph Heller","year": 1961,
        "characters": ["John Yossarian", "Captain Aardvark",
        "Chaplain Tappman", "Colonel Cathcart",
        "Doctor Daneeka"],"tags": ["novel"],
        "copies": 6, "available" : false}
    } ]
  }
}
```

这表明两种形式是等效的。其实不对，因为它们应用过滤和搜索的顺序是不同的。在第一种情况下，过滤器应用到查询所发现的所有文档上。第二种情况下，过滤发生在在运行查询之前，性能更好。如前所述，过滤器很快，所以filtered查询效率更高。6.2节将讨论这些内容。

3.5.2　过滤器类型

我们现在已经知道如何使用过滤器，也知道两种过滤方法的区别。现在看看Elasticsearch提

供的过滤器类型。

1. 范围过滤器

范围过滤器可以用来限制只搜索那些字段值在给定边界之间的文档。例如，为了创建一个过滤器来过滤只在1930~1990年之间出版的图书，在查询中加入以下部分：

```
{
  "post_filter" : {
    "range" : {
      "year" : {
        "gte": 1930,
        "lte": 1990
      }
    }
  }
}
```

使用gte和lte，表明字段的左右边界是包含的。如果想排除左右边界，可以使用gt和lt参数。如果想要从1930年（含1930）到1990年（不含）的文档，构造下列过滤器：

```
{
  "post_filter" : {
    "range" : {
      "year" : {
        "gte": 1930,
        "lt": 1990
      }
    }
  }
}
```

总结如下。

- gt：大于。
- lt：小于。
- gte：大于或等于。
- lte：小于或等于。

你也可以使用execution参数。这是一个对引擎如何执行过滤器的提示，可用值有fielddata和index。一般的经验是：在范围内有很多值时，fielddata能提高性能（以及内存使用）；范围内的值较少时，index应该更好。

还有个过滤器的变种：numeric_filter。这是一个特别设计的版本，用来过滤数值类型的值。此过滤器更快，但有额外的内存使用，因为Elasticsearch需要加载过滤字段的值。请注意，有时即使不使用范围过滤器，这些值也被加载。这种情况发生在我们使用相同字段做切面或排序时，没有理由不使用此过滤器。

2. exists过滤器

exists过滤器非常简单。它过滤掉给定字段上没有值的文档。比如，考虑下面的代码：

```
{
  "post_filter" : {
    "exists" : { "field": "year" }
  }
}
```

上述过滤器将只返回year字段有值的文档。

3. missing过滤器

missing过滤器跟exists过滤器相反，它过滤掉给定字段上有值的文档。然而，它还有一些额外的功能。除了选择指定字段缺失的文档，可以指定Elasticsearch对空字段的定义。这有助于在输入数据中包含null、EMPTY、not-defined等词条的情况。我们修改先前的例子，找没有定义year字段、或者year字段为0的那些文档。修改过的过滤器将如下所示：

```
{
  "post_filter" : {
    "missing" : {
      "field": "year",
      "null_value": 0,
      "existence": true
    }
  }
}
```

在前面的示例中，有两个额外参数。existence参数告诉Elasticsearch应该检查指定字段上存在值的文档，null_value参数定义了应该被视为空的额外值。如果你没有定义null_value，existence将被设为默认值；所以，在这个例子中可以省略existence。

4. 脚本过滤器

有时，我们想要通过计算值来过滤文档。一个例子是：可以过滤掉所有发表在一个世纪以前的书。使用脚本过滤器来实现，如下所示：

```
{
  "post_filter" : {
    "script" : {
      "script" : "now - doc['year'].value > 100",
        "params" : {
          "now" : 2012
        }
      }
    }
  }
}
```

可以看到，我们使用了一个简单的脚本来计算值，并从中过滤数据。5.2节将讨论Elasticsearch

的更多脚本功能。

5. 类型过滤器

类型过滤器专门用来限制文档的类型。当查询运行在多个索引上，或单个索引但有很多类型时，可以使用这个过滤器。例如，想限制返回文档的类型为book，使用下列过滤器：

```
{
  "post_filter" : {
    "type": {
     "value" : "book"
    }
  }
}
```

6. 限定过滤器

限定过滤器限定单个分片返回的文档数目。不要把它跟size参数混在一起。作为例子，我们看看下面的过滤器：

```
{
  "post_filter" : {
    "limit" : {
      "value" : 1
    }
  }
}
```

当我们对分片数量使用默认设置时，上述过滤器返回5个文档。因为在Elasticsearch中，索引默认分为5个分片。查询分别运行在各个分片上，而每个分片最多返回1个文档。

7. 标识符过滤器

需要过滤成若干具体的文档时，可以使用标识符过滤器。例如，要排除标识符等于1的文档，可使用类似下面的代码：

```
{
  "post_filter": {
    "ids" : {
      "type": ["book"],
      "values": [1]
    }
  }
}
```

注意，type参数不是必需的。然而，当我们在几个索引中搜索，又想指定一个感兴趣的类型时，它还是有用的。

8. 如果还不够

目前为止，我们讨论了几个在Elasticsearch使用过滤器的例子。然而，这只是冰山一角。你

可以在过滤器中封装几乎所有查询。例如，让我们来看看下面的查询：

```
{
  "query" : {
    "multi_match" : {
      "query" : "novel erich",
      "fields" : [ "tags", "author" ]
    }
  }
}
```

上述示例显示了一个我们熟知的简单multi_match查询。可以用过滤器按如下方式重写此查询：

```
{
  "post_filter" : {
    "query" : {
      "multi_match" : {
        "query" : "novel erich",
        "fields" : [ "tags", "author" ]
      }
    }
  }
}
```

当然，唯一的区别是得分。过滤器返回的每个文档得分都是1.0。注意，Elasticsearch有几个专用过滤器是这样工作的（比如与词条查询对应的词条过滤器）。所以，你不必总是使用封装查询语法。事实上，你应该尽量使用专用的过滤器版本。

Elasticsearch支持下列专用过滤器：

- bool过滤器；
- geo_shape过滤器；
- has_child过滤器；
- has_parent过滤器；
- ids过滤器；
- indices过滤器；
- match_all过滤器；
- nested过滤器；
- prefix过滤器；
- range过滤器；
- regexp过滤器；
- term过滤器；
- terms过滤器。

9. 组合过滤器

现在，是时候把一些过滤器组合在一起了。第一个选择是使用bool过滤器，它能够在3.4.1节所述原理的基础上组合过滤器。第二种选择是使用and、or和not过滤器。and过滤器使用一个过滤器数组并返回与该数组中的所有过滤器匹配的文档。or过滤器也使用一个数组，但它返回与数组中任何一个过滤器匹配的文档。至于not过滤器，则返回与所封装的过滤器不匹配的文档。当然，所有这些过滤器可以嵌套使用，如以下示例所示：

```
{
  "post_filter": {
    "not": {
      "and": [
        {
          "term": {
            "title": "Catch-22"
          }
        },
        {
          "or": [
            {
              "range": {
                "year": {
                  "gte": 1930,
                  "lte": 1990
                }
              }
            },
            {
              "term": {
                "available": true
              }
            }
          ]
        }
      ]
    }
  }
}
```

- 关于bool过滤器

当然，你可能会问bool过滤器与and、or、not过滤器之间的区别。首先，这些过滤器可以互换使用。当然，从返回的结果角度是对的，但从性能角度则不然。

我们可以看到Elasticsearch在内部为每个过滤器都建立了一个叫bitset的结构，它保存着索引中的后续文档是否跟过滤器匹配的信息。bitset很容易被缓存并在使用相同过滤器的所有查询中重用。这是Elasticsearch的一种简便、高效的任务。总之，尽可能使用bool过滤器。可惜，现实生活不总是这么简单。某些类型的过滤器没有能力直接创建bitset。在这罕见的情形下，bool筛选

器将更低效。你已经知道有两个这样的过滤器：数值型的范围过滤器和脚本过滤器。第三个是使用地理坐标的整组过滤器，6.2.10将对此进行讨论。

10. 命名过滤器

设置过滤器可能很复杂，所以有时候，如果知道哪些过滤器用来决定查询应该返回哪些文档，无疑是很有帮助的。幸好，可以给每个过滤器命名，名字将随着匹配文档返回。来看看它是如何工作的。以下查询将返回所有可用并且标签为`novel`，或者来自19世纪的书：

```
{
  "query": {
    "filtered" : {
      "query": { "match_all" : {} },
      "filter" : {
        "or" : [
          { "and" : [
            { "term": { "available" : true } },
            { "term": { "tags" : "novel" } }
            ]},
          { "range" : { "year" : { "gte": 1800, "lte" : 1899 } } }
        ]
      }
    }
  }
}
```

我们使用查询的`filtered`版本，因为它是Elasticsearch中唯一可以添加过滤器信息的版本。我们重写该查询以添加每个过滤器的名字，如下所示：

```
{
  "query": {
    "filtered" : {
      "query": { "match_all" : {} },
      "filter" : {
        "or" : {
          "filters" : [
            {
              "and" : {
              "filters" : [
                {
                 "term": {
                  "available" : true,
                  "_name" : "avail"
                  }
                },
                {
                  "term": {
                    "tags" : "novel",
                    "_name" : "tag"
                  }
                }
              ],
```

```
                  "_name" : "and"
                  }
                },
                {
                  "range" : {
                    "year" : {
                      "gte": 1800,
                    "lte" : 1899
                  },
                  "_name" : "year"
                }
              }
            ],
            "_name" : "or"
          }
          }
        }
      }
    }
  }
```

可以看到，我们在每个过滤器中添加了_name属性。对于and和or过滤器，需要改变语法。所以，用额外的对象封装了所附过滤器，以得到正确的JSON格式。发送此查询，应该得到类似下面的响应：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title": "All Quiet on the
        Western Front","otitle": "Im Westen nichts Neues",
        "author": "Erich Maria Remarque","year": 1929,
        "characters": ["Paul Bäumer", "Albert Kropp",
        "Haie Westhus", "Fredrich Müller",
        "Stanislaus Katczinsky", "Tjaden"],
        "tags": ["novel"],"copies": 1, "available": true},
      "matched_queries" : [ "or", "tag", "avail", "and" ]
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 1.0, "_source" : { "title": "Crime and
```

```
          Punishment","otitle": "Преступлéние и наказáние",
          "author": "Fyodor Dostoevsky","year": 1886,
          "characters": ["Raskolnikov", "Sofia Semyonovna
          Marmeladova"],"tags": [],"copies": 0, "available" : true},
        "matched_queries" : [ "or", "year", "avail" ]
      } ]
    }
}
```

你可以看到，除了标准信息以外，每个文档包含一个表，包含与该特定文档匹配的过滤器名称。

> 记住在大多数情况下，filtered查询比post_filter更快。所以在可能的情况下，尽可能使用filtered查询。

3.5.3　过滤器的缓存

关于过滤器最后要提到的是缓存。缓存加速了使用过滤器的查询，代价是第一次执行过滤器时的内存成本和查询时间。因此，缓存的最佳选择是那些可以重复使用的过滤器，例如，经常会使用并包括参数值的那些。

缓存可以在and、bool、or过滤器上打开（但通常，缓存它们所附的过滤器才是更好的主意）。在这种情况下，所需的语法与前面命名过滤器所描述的一样，如下所示：

```
{
  "post_filter" : {
    "script" : {
      "_cache": true,
      "script" : "now - doc['year'].value > 100",
      "params" : {
        "now" : 2012
      }
    }
  }
}
```

有些过滤器不支持_cache参数，因为它们的结果总是被缓存。默认情况下是下面这些：

- exists
- missing
- range
- term
- terms

可通过关闭缓存来修改此行为，代码如下所示：

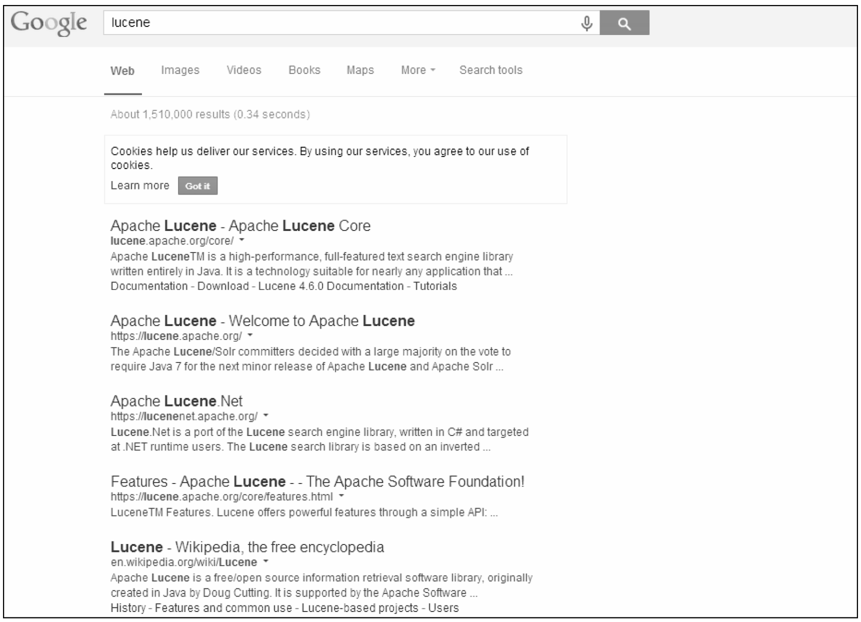
```
{
  "post_filter": {
```

```
      "term": {
        "_cache": false,
        "year": 1961
      }
    }
  }
```

对于ids过滤器、match_all过滤器和limit过滤器，缓存是无效的。

3.6 高亮显示

你可能听说过高亮显示，即使不熟悉这个名字，也可能在你访问过的网页中看过高亮显示的结果。高亮显示是在结果文档中显示查询中的哪个或哪些单词被匹配的过程。例如，在谷歌搜索lucene这个词，我们会看到它在结果列表中以粗体显示，如下面的截图所示：



本章将展示如何使用Elasticsearch高亮能力来加强我们的应用，使结果高亮显示。

3.6.1 高亮显示入门

要展示高亮显示是如何工作的，最好的办法是创建一个查询，看看Elasticsearch返回的结果。所以假设我们想高亮显示在title字段中匹配的单词，以改善用户的搜索体验。我们还是来搜索crime一词，为了得到具有高亮的结果，发送如下查询：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "fields" : {
      "title" : {}
    }
  }
}
```

这个查询的响应看起来如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.19178301, "_source" : { "title": "Crime and
        Punishment","otitle": "Преступлéние и наказáние",
        "author": "Fyodor Dostoevsky","year": 1886,

        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"],"tags": [],"copies": 0, "available" : true},
      "highlight" : {
        "title" : [ "<em>Crime</em> and Punishment" ]
      }
    } ]
  }
}
```

可以看到，除了从Elasticsearch得到的标准信息，有一个新的名为`highlight`的部分。Elasticsearch使用`<em>`这个HTML标签来包含高亮部分。这是Elasticsearch的默认行为，我们将学习如何去改变它。

3.6.2　字段配置

为了执行高亮显示，需要呈现字段的原始内容：我们必须把这些用来高亮显示的字段设为`stored`，或者把这些字段包含在`_source`字段中。

3.6.3 深入底层

Elasticsearch在底层使用Apache Lucene，而高亮显示是Lucene库的功能之一。Lucene提供了三种类型的高亮实现：标准类型，就是我们刚刚使用的；第二种叫FastVectorHighlighter，它需要词向量和位置才能工作；第三种叫PostingsHighlighter，本章的最后将讨论它。Elasticsearch自动选择正确的高亮实现方式：如果字段的配置中，term_vector属性设成了with_positions_offsets，则将使用FastVectorHighlighter。

然而，必须记住，使用词向量将导致索引变大，但高亮显示的执行需要更少的时间。此外，对于存储了大量数据的字段来说，推荐使用FastVectorHighlighter

3.6.4 配置 HTML 标签

我们已经提到，改变默认的HTML标签成我们想用的标签是可能的。例如，假设要使用标准的HTML标签来高亮。为此，我们应分别设置pre_tags和post_tags这些属性（它们是数组）为和。既然提到的两个属性为数组，可以包含多个标签，Elasticsearch将使用定义的每个标签来高亮显示不同的单词。所以，我们的示例查询如下所示：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "pre_tags" : [ "<b>" ],
    "post_tags" : [ "</b>" ],
    "fields" : {
      "title" : {}
    }
  }
}
```

对上述查询，Elasticsearch返回的结果如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
```

```
        "_index" : "library",
        "_type" : "book",
        "_id" : "4",
        "_score" : 0.19178301, "_source" : { "title": "Crime and
          Punishment","otitle": "Преступлéние и наказáние",
          "author": "Fyodor Dostoevsky","year": 1886,
          "characters": ["Raskolnikov", "Sofia Semyonovna
          Marmeladova"],"tags": [],"copies": 0, "available" : true},
        "highlight" : {
          "title" : [ "<b>Crime</b> and Punishment" ]
        }
      } ]
    }
  }
```

可以看到，title字段中的Crime，被我们选择的标签包围。

3.6.5 控制高亮片段

Elasticsearch允许我们控制高亮片段的数量以及它们的大小，为此公开了两个属性供使用。第一个，number_of_fragments，定义Elasticsearch返回的片段数量，默认值为5。把这个属性设置为0，将导致整个字段被返回，这对短字段来说是很便利的；然而，对长字段来说代价较大。第二个属性是fragment_size，用来指定高亮片段的最大字符长度，默认值是100。

3.6.6 全局设置与局部设置

前面讨论的高亮显示的属性，可以设在全局范围，也可以设在每个字段上。全局设置将用在没有做局部设置的所有字段上，它跟高亮对象的fields节点设在同一个级别上，如下所示：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "pre_tags" : [ "<b>" ],
    "post_tags" : [ "</b>" ],
    "fields" : {
      "title" : {}
    }
  }
}
```

也可以为每个字段设置这些属性。比如，除了title字段，我们想对其他字段保持默认行为，可以使用下面的代码：

```
{
  "query" : {
    "term" : {
```

```
      "title" : "crime"
    }
  },
  "highlight" : {
    "fields" : {
      "title" : {
        "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ]
      }
    }
  }
}
```

可以看到，我们把它放在指定title字段行为的空JSON对象里面，而不是与fields同一水平线上的部分中。当然，每个字段可配置为不同的属性。

3.6.7　需要匹配

有时，尤其是在使用多个高亮字段时，只需要显示跟查询匹配的字段。为了触发此行为，要把require_field_match属性设为true。把该属性设为false将导致所有词条都高亮显示，即使是在跟查询不匹配的字段中。

为了看它是如何工作的，创建一个新的索引users，并创建一个文档到这个索引中，发送如下命令：

```
curl -XPUT 'http://localhost:9200/users/user/1' -d '{
  "name" : "Test user",
  "description" : "Test document"
}'
```

现在，假设想高亮显示name和description字段，查询如下所示：

```
{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ]
        },
      "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
        "</b>" ] }
    }
  }
}
```

上述查询的结果如下所示：

```
{
  "took" : 3,
  "timed_out" : false,
```

```
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 0.19178301,
      "hits" : [ {
        "_index" : "users",
        "_type" : "user",
        "_id" : "1",
        "_score" : 0.19178301, "_source" : {"name" : "Test
          user","description" : "Test document"},
        "highlight" : {
        "description" : [ "<b>Test</b> document" ],
        "name" : [ "<b>Test</b> user" ]
      }
    } ]
  }
}
```

注意，即使我们只匹配name字段，得到的结果却高亮显示了上述两个字段。在大多数情况下，我们不希望这样。所以现在修改查询，使用require_field_match属性如下：

```
{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "require_field_match" : "true",
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "<b>" ]
        },
      "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
        "</b>" ] }
    }
  }
}
```

看看修改过的查询得到的结果，如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
```

```
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "users",
      "_type" : "user",
      "_id" : "1",
      "_score" : 0.19178301, "_source" : {"name" : "Test
        user","description" : "Test document"},
      "highlight" : {
        "name" : [ "<b>Test</b> user" ]
      }
    } ]
  }
}
```

可以看到，Elasticsearch只在高亮中返回匹配字段，在我们的例子中是name字段。

3.6.8　信息高亮器

现在来讨论一下Elasticsearch中的第三种高亮显示类型。它在Elasticsearch 0.90.6中被加入，与前两种类型略有不同，我们通过下面的例子来看看这些区别。当字段定义中的index_options属性设成offsets时，自动运用PostingsHighlighter。所以，为了演示PostingsHighlighter是如何工作的，我们将用正确的映射创建一个简单的索引。为此，执行以下命令：

```
curl -XPUT 'localhost:9200/hl_test'
curl -XPOST 'localhost:9200/hl_test/doc/_mapping' -d '{
  "doc" : {
    "properties" : {
      "contents" : {
        "type" : "string",
        "fields" : {
          "ps" : { "type" : "string", "index_options" : "offsets" }
        }
      }
    }
  }
},
```

> 请记住，与FastVectorHighlighter类似，PostingsHighlighter需要的offset会导致索引大小的增加，但这种增加比使用词向量时更小。此外，索引offset比索引词向量更快，且PostingsHighlighter的查询性能更好。

如果一切顺利，我们将有一个新的索引和映射。映射定义了两个字段：一个名叫contents，另一个叫contents.ps。在这个例子中，使用index_options属性打开了偏移。这意味着Elasticsearch将对contents contents.ps字段使用信息高亮器。

为了看其中的差别，我们索引一个文档，该文档包含维基百字段使用标准的高亮类型，而对

科中描述伯明翰历史的片段。为此，执行以下命令：

```
curl -XPUT localhost:9200/hl_test/doc/1 -d '{
  "contents" : "Birmingham''s early history is that of a remote and
    marginal area. The main centers of population, power and wealth
    in the pre-industrial English Midlands lay in the fertile and
    accessible river valleys of the Trent, the Severn and the Avon.
    The area of modern Birmingham lay in between, on the upland
    Birmingham Plateau and within the densely wooded and sparsely
    populated Forest of Arden."
}'
```

最后一步是使用两个高亮类型来发送查询请求。为此，可以使用如下命令来发送单个请求：

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{
  "query": {
    "term": {
       "contents": "modern"
    }
  },
  "highlight": {
    "fields": {
       "contents": {},
       "contents.ps" : {}
    }
  }
}'
```

如果一切顺利，可以在响应中找到如下片段：

```
"highlight" : {
  "contents" : [ " valleys of the Trent, the Severn and the
    Avon. The area of <em>modern</em> Birmingham lay in
    between, on the upland" ],
  "contents.ps" : [ "The area of <em>modern</em> Birmingham lay
    in between, on the upland Birmingham Plateau and within the
    densely wooded and sparsely populated Forest of Arden." ]
}
```

可以看到，两种高亮实现都找到了所需要的单词，不同的是，信息高亮器返回的片段更聪明，它检测了句子的边界。

用下面的命令再来试一个查询：

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{
  "query": {
    "match_phrase": {
      "contents": "centers of"
    }
  },
  "highlight": {
    "fields": {
      "contents": {},
```

```
        "contents.ps": {}
      }
    }
  }'
```

搜索一个特定的短语centers of。正如你预料的那样，这两种高亮实现的结果不一样。对标准高亮实现，将在响应中找到如下的短语：

```
"Birmingham's early history is that of a remote and marginal area.
  The main <em>centers</em> <em>of</em> population"
```

可以清楚地看到，标准高亮实现分割了给定短语，高亮单独的词条。不是所有的centers和of词条都被高亮，只有来自这个短语的才被高亮。

3

另一方面，信息高亮器返回如下高亮片段：

```
"Birmingham's early history is that <em>of</em> a remote and marginal
  area.",
"The main <em>centers</em> <em>of</em> population, power and wealth
  in the pre-industrial English Midlands lay in the fertile and
  accessible river valleys <em>of</em> the Trent, the Severn and the
  Avon.",
"The area <em>of</em> modern Birmingham lay in between, on the upland
  Birmingham Plateau and within the densely wooded and sparsely
  populated Forest <em>of</em> Arden."
```

这是个显著的差异：信息高亮器高亮了所有跟查询词条匹配的词条，而不仅是组成短语的那些词条。

3.7　验证查询

有时，应用程序发送到Elasticsearch的查询是自动从多个条件生成的，甚至更糟，它们是通过某种向导生成，最终用户可以在其中创建复杂的查询。问题是，查询的正确与否，有时并不容易分辨。为了解决这个问题，Elasticsearch公开了验证API。

使用验证 API

验证API非常简单，我们把它发送到_validate_query端点，而不是_search端点就行了。来看看下面的查询：

```
{
  "query" : {
    "bool" : {
      "must" : {
        "term" : {
          "title" : "crime"
```

```
        }
      },
      "should" : {
        "range : {
          "year" : {
            "from" : 1900,
            "to" : 2000
          }
        }
      },
      "must_not" : {
        "term" : {
          "otitle" : "nothing"
        }
      }
    }
  }
}
```

本书中已经用过这个查询，我们知道该查询一切正常。但要通过下面的命令来验证一下（已经把查询保存到query.json文件中）：

```
curl -XGET 'localhost:9200/library/_validate/query?pretty' -d
  @query.json
```

查询看上去是对的，但来看看验证API怎么说。Elasticsearch返回的响应如下所示：

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

看下valid属性，它被设成了false。有些地方出错了。再执行一次验证查询，这次加入explain参数，如下所示：

```
curl -XGET 'localhost:9200/library/_validate/query?pretty&explain' --
  data-binary @query.json
```

这次，Elasticsearch返回的结果更加详细，如下所示：

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "explanations" : [ {
    "index" : "library",
```

```
    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParsingException:
      [library] Failed to parse;
      org.elasticsearch.common.jackson.core.JsonParseException:
      Illegal unquoted character ((CTRL-CHAR, code 10)): has to be
      escaped using backslash to be included in name\n at [Source:
      [B@6456919f; line: 10, column: 18]"
  } ]
}
```

现在一切都清楚了，在我们的例子中，range属性的引号少了一个。

> 你可能想知道为什么我们在curl查询中使用--data-binary参数。此参数可在发送查询到Elasticsearch时保留换行符。这意味着行列数都将是完好的，这样更容易发现错误。在其他情况下，-d参数更为方便，因为它更短。

验证API还可以检测其他错误，例如，格式不正确的数字或其他映射相关的问题。可惜，由于我们的应用程因为缺乏错误信息中的结构，不是很容易发现问题。

3.8 数据排序

我们现在知道了如何构建查询和过滤结果，也知道搜索类型以及为什么它们重要。可以把这些查询发送到Elasticsearch，并分析返回的数据分析。现在，这个数据的组织顺序是由得分决定的。在大多数情况下，这正是我们想要的。搜索操作首先应该给我们最相关的文档。然而，如果想让查询更像一个数据库，或想设置一个更复杂的算法对数据排序，该怎么做呢？来看看Elasticsearch的排序功能可以做什么。

3.8.1 默认排序

看下面的查询，它返回至少含有一个指定单词的所有书：

```
{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  }
}
```

在底层，Elasticsearch把它当作如下查询：

```
{
  "query" : {
    "terms" : {
```

```
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  },
  "sort" : { "_score" : "desc" }
}
```

注意上述查询中高亮的部分，这是Elasticsearch的默认排序。更详细来说，这个片段如下所示：

```
"sort" : [
  { "_score" : "desc" }
]
```

前一节定义了在结果列表中的文档应该如何排序。在这个例子中，Elasticsearch在结果列表的顶部显示最高分的文档。最简单的修改是旋转`sort`部分，以达到反向排序，如下所示：

```
"sort" : [
  { "_score" : "asc" }
]
```

3.8.2　选择用于排序的字段

默认排序很无聊，不是吗？所以，我们让它根据文档中的一个字段排序，如下所示：

```
"sort" : [
  { "title" : "asc" }
]
```

可惜，这并不会如预期一样工作。虽然Elasticsearch对文档做了排序，但文档顺序有些奇怪。仔细查看响应，Elasticsearch对每个文档返回了排序信息，例如，对`Catch-22`这本书，返回文档类似于以下代码：

```
{
  "_index": "library",
  "_type": "book",
  "_id": "2",
  "_score": null,
  "_source": {
    "title": "Catch-22",
    "author": "Joseph Heller",
    "year": 1961,
    "characters": [
       "John Yossarian",
       "Captain Aardvark",
       "Chaplain Tappman",
       "Colonel Cathcart",
       "Doctor Daneeka"
    ],
    "tags": [
       "novel"
    ],
```

```
        "copies": 6,
        "available": false,
        "section": 1
    },
    "sort": [
        "22"
    ]
}
```

如果你比较一下title字段和返回的排序信息，一切应该都清楚了。Elasticsearch在分析过程中把字段拆分为几个标记。因为排序是使用单个标记，Elasticsearch在产生的标记中选择了一个。这种做法是最好的，因为它可以通过对这些标记按照字母顺序排序并选择了第一个。这就是为什么在排序的值中，我们只能找到一个单词而不是title字段的全部内容。在空余时间，你可以看看Elasticsearch对字符字段排序时的表现。

一般来说，对一个未经分析的字段排序是一个好主意。我们可以对具有多个值的字段排序，但在大多数情况下，没有多大意义，因此使用有限。例如，我们使用两个不同的字段，一个作为排序，另一个作为搜索，修改title字段。更改过的title字段定义可能类似于下面的代码：

```
"title" : {
  "type": "string",
  "fields": {
    "sort": { "type" : "string", "index": "not_analyzed" }
  }
}
```

在映射中修改title字段后，本章开头已经展示过，可以尝试对title.sort字段排序，看看它是否工作。为此，需要发送以下查询：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : [
    {"title.sort" : "asc" }
  ]
}
```

现在，它正常了。正如你所看到的，我们用新的字段title.sort。已经将其设置为未经分析，因此，在索引中它是个单值字段。

在Elasticsearch响应中，每个文档包含用于排序的值的有关信息，如下所示：

```
"_index" : "library",
"_type" : "book",
"_id" : "1",
"_score" : null, "_source" : { "title": "All Quiet on the
  Western Front","otitle": "Im Westen nichts Neues",
  "author": "Erich Maria Remarque","year":
  1929,"characters": ["Paul Bäumer", "Albert Kropp",
  "Haie Westhus", "Fredrich Müller", "Stanislaus
  Katczinsky", "Tjaden"],"tags": ["novel"],"copies": 1,
```

```
    "available": true, "section" : 3},
  "sort" : [ "All Quiet on the Western Front" ]
```

请注意，sort在请求和响应中，都是一个数组。这表明我们可以使用几种不同的排序。对于前一个字段值相同的文档，Elasticsearch将使用下一个数组里的元素来确定文档的顺序。所以，如果文档的title字段相同，将使用我们指定的下一个字段排序。

3.8.3　指定缺少字段的行为

当有些与查询匹配的文档没有我们要排序的字段时，会怎么样？默认情况下，没有给定字段的文档，如果是升序排，则会出现在第一个；如果是降序排，则出现在最后一个。然而，有时这不是我们想要的。

使用数字字段排序时，可以更改Elasticsearch对缺少字段的文档的默认行为。例如以下查询：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : [
    { "section" : { "order" : "asc", "missing" : "_last" } }
  ]
}
```

注意，查询中sort节点的扩展部分添加了missing参数。通过把missing参数设为_last，Elasticsearch将把缺乏给定字段的文档放在结果列表的底部。设置为_first，则会把缺乏给定字段的文档放在结果列表的顶部。值得一提的是，除了_last和_first值，Elasticsearch允许我们使用任意数字。在这种情况下，一个没有给定字段的文档将被视为该文档具有给定的值。

3.8.4　动态条件

上一节提到，Elasticsearch允许使用具有多个值的字段排序。可以使用脚本来控制排序时进行的比较，通过告诉Elasticsearch如何计算应用于排序的值来达到目的。假设要通过tags字段中的第一个值来排序。看看下面的示例查询：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc['tags'].values.length > 0 ?
        doc['tags'].values[0] : '\u19999'",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

在上面的示例中，我们把每一个不存在的值替换成一个Unicode字符，该字符在列表中应该处于足够低的位置。此代码的主要想法是检查tags数组中是否包含至少一个元素。如果是，那么返回数组中的第一个值。如果数组为空，返回Unicode字符，该字符应该放在结果列表的底部。除了script参数，还需要指定order参数（在我们的例子中是升序），以及用于比较的type参数（我们的脚本返回的是string）。

3.8.5 排序规则和国家特有字符

如果想使用英语以外的语言，可能要面对字符顺序不正确的问题。这是因为许多语言有不同的字母顺序定义。Elasticsearch支持多种语言，但需要额外的插件来支持适当的排序规则。它很容易安装和配置，8.7节将进一步讨论。

3.9 查询重写

基本上，任何涉及多词条的查询，比如前缀查询和通配符查询，都使用查询重写。Elasticsearch这样做是基于性能方面的原因。重写过程把原始的、昂贵的查询修改成一组Lucene认为不太昂贵的查询。

3.9.1 重写过程示例

要说明重写过程在内部是如何进行的，最好的方式是看一个例子，看看哪些词条代替了原始的查询词条。假设在索引中有以下数据：

```
curl -XPOST 'localhost:9200/library/book/1' -d '{"title": "Solr 4
  Cookbook"}'
curl -XPOST 'localhost:9200/library/book/2' -d '{"title": "Solr 3.1
  Cookbook"}'
curl -XPOST 'localhost:9200/library/book/3' -d '{"title": "Mastering
  Elasticsearch"}'
```

我们需要找到以字母s开头的所有文档。就这么简单，对该library索引执行以下查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "prefix" : {
      "title" : "s",
      "rewrite" : "constant_score_boolean"
    }
  }
}'
```

这里，使用一个简单的前缀查询。我们说过要找到所有在title字段中含有字母s的文档。我们还使用了rewrite属性来指定查询重写方法，但先跳过它，因为会在本节的后半部分讨论此参数的可能值。作为对上述查询的响应，得到以下输出：

```
{
  "took" : 22,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"title": "Solr 3.1 Cookbook"}
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"title": "Solr 4 Cookbook"}
    } ]
  }
}
```

可以看到，在响应中，我们有两个文档，它们的标题字段内容以期望的字符开头。看看Lucene级别的查询，我们注意到，前缀查询被重写成类似于下面这样的查询：

```
ConstantScore(title:solr)
```

这是因为solr是唯一以字母s开头的词条。这就是查询重写的一切：找到相关的词条，把查询重写为性能更好的查询，而不是执行昂贵的查询。

3.9.2　查询重写的属性

我们已经说过，可以在任何多项词条查询（比如Elasticsearch的前缀查询和通配符查询）中使用rewrite参数来控制查询如何被改写。把rewrite参数添加到负责实际查询的JSON对象中，如下所示：

```
{
  "query" : {
    "prefix" : {
      "title" : "s",
      "rewrite" : "constant_score_boolean"
    }
  }
}
```

现在，来看看此参数的值有哪些选项。

- scoring_boolean：这种重写方法把生成的每个词条翻译成布尔查询中的一个should子

句。此查询重写方法可能是CPU密集型（因为它计算并存储每个词条的得分），如果查询许多词条，可能超过布尔查询极限，也就是1024。此外，此查询会存储计算所得的分数。

- constant_score_boolean：这种重写方法类似于上面描述的scoring_boolean重写方法，但是对CPU要求较低，因为不需要计算得分。相反，每个词条都得到一个与查询加权相等的得分，默认是1，可以通过加权属性进行设置。与scoring_boolean重写方法类似，该方法也可能达到布尔查询的最高限制。
- constant_score_filter：就像Apache Lucene的Javadocs声明的那样，这个重写方法按顺序访问每个词条，标记该词条的所有文档，并创建一个私有过滤器来重写查询。匹配的文档都被赋予一个与查询加权相等的常量得分。当匹配词条或文档的数量很大时，此方法比scoring_boolean和constant_score_boolean快。
- top_terms_N：这种重写方法把生成的每个词条翻译成布尔查询中的一个should子句，并保持查询计算所得的分数。然而，与scoring_boolean重写方法不同，它只会保留N个最高得分的词条，以免达到布尔查询的最大限制。
- top_terms_boost_N：这是一种类似top_terms_N的重写方法。然而，与top_terms_N重写方法不同，分数只由加权计算而来，而非查询。

> 当重写属性设置为constant_score_auto或根本没有设置时，根据查询以及构造方式的不同，将选择使用constant_score_filter或constant_score_boolean。

结束本章查询重写部分之前，我们应该问自己最后一个问题，“什么时候使用哪种类型的重写”？这个问题的答案在很大程度上取决于我们的用例，但是总结来说，如果能忍受低精度（但性能更好），使用top N重写方法。如果需要高精度（但性能较低），选择布尔方法。

3.10　小结

本章介绍了Elasticsearch查询如何工作，以及如何选择要返回的数据；讨论了查询重写如何工作，有哪些搜索类型，什么是搜索偏好；展示了Elasticsearch中可用的基本查询，并使用过滤器来过滤结果。此外，还讨论了高亮显示功能，让它来高亮文档中匹配的部分。我们验证了查询。了解了复合查询，将多个查询组合在一起，最后，看到了如何根据需求配置排序。

下一章重点介绍索引，但还会涉及其他内容。我们会学到如何索引树状结构；看到如何在Elasticsearch中存储JSON对象，来索引非扁平的数据，以及如何修改一个已经创建的索引的结构。下一章还将阐述如何使用嵌套文档和主从功能来处理文档之间的关系。

第 4 章 扩展索引结构

上一章介绍了不少有关查询Elasticsearch的知识，展示了如何在Elasticsearch中选择返回字段，并说明了查询的工作方式。除此之外，我们还了解到可用的基本查询以及过滤数据的方法，如何高亮文档中的匹配文字，如何验证查询、复合查询，以及如何对数据排序。这一章的主要内容如下：

- 索引树型结构数据；
- 索引非扁平数据；
- 在可能情况下修改索引结构；
- 使用嵌套文档索引关系型数据；
- 使用主从功能索引关系型数据。

4.1 索引树形结构

树型结构随处可见。如果开发一个商店应用程序，可能会需要类别。看看文件系统，文件和目录以树状结构排列。本书也呈树型：章节包含各种主题，而主题又划分为副主题。想象得出，Elasticsearch也能够索引树状结构。让我们看看如何通过path_analyzer浏览这种数据类型。

4.1.1 数据结构

首先，通过以下代码创建一个简单的索引结构：

```
curl -XPUT 'localhost:9200/path' -d '{
  "settings" : {
    "index" : {
      "analysis" : {
        "analyzer" : {
          "path_analyzer" : { "tokenizer" : "path_hierarchy" }
        }
      }
    }
  },
```

```
  "mappings" : {
    "category" : {
      "properties" : {
        "category" : {
          "type" : "string",
          "fields" : {
            "name" : { "type" : "string",
                       "index" : "not_analyzed" },
            "path" : { "type" : "string",
                       "analyzer" : "path_analyzer",
                       "store" : true }
          }
        }
      }
    }
  }
}'
```

可以看到，我们创建了一个类型：`category`。我们将使用它在树型结构中存储文档位置的信息。想法很简单，可以用与在硬盘里显示文件和目录完全相同的方式，以路径的形式显示文档位置。例如，在一家汽车店中可以有如下路径：/cars/passenger/sport、/cars/passenger/camper，或者/cars/delivery_truck/。需要用三种方式对这个路径建立索引。我们将使用未经额外处理的`name`字段，以及一个使用定义的`path_analyzer`处理的`path`字段，也保留原始值以方便搜索。

4

4.1.2 分析

现在，让我们看看Elasticsearch在分析的过程中如何处理类别路径。为此，运用5.7节中描述的分析API，执行以下命令：

```
curl -XGET 'localhost:9200/path/_analyze?field=category.path&pretty' -d
'/cars/passenger/sport'
```

Elasticsearch返回的结果如下所示：

```
{
  "tokens" : [ {
    "token" : "/cars",
    "start_offset" : 0,
    "end_offset" : 5,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "/cars/passenger",
    "start_offset" : 0,
    "end_offset" : 15,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "/cars/passenger/sport",
```

```
      "start_offset" : 0,
      "end_offset" : 21,
      "type" : "word",
      "position" : 1
    } ]
}
```

可以看到，Elasticsearch把类别路径/cars/passenger/sport处理并分解成三个标记。归功于此，我们很容易通过词条过滤器找到每个属于指定类别或子类别的文档。举例如下：

```
{
  "filter" : {
   "term" : { "category.path" : "/cars" }
  }
}
```

注意，我们还在索引中建立了`category.name`字段的原始值，便于直接找到特定路径的文档，略过层次结构更深的文档。

4.2　索引非扁平数据

并非所有数据都是扁平数据，就像到目前为止本书使用的所有数据。如果我们构建使用Elasticsearch的系统，应创建有利于Elasticsearch的结构。结构不总是扁平的，因为并非所有用例都允许这样操作。让我们看看如何使用完全结构化的JSON对象创建映射。

4.2.1　数据

假定有如下数据（存储于structured_data.json文件中）：

```
{
  "book" : {
    "author" : {
      "name" : {
        "firstName" : "Fyodor",
        "lastName" : "Dostoevsky"
      }
    },
    "isbn" : "123456789",
    "englishTitle" : "Crime and Punishment",
    "year" : 1886,
    "characters" : [
      {
        "name" : "Raskolnikov"
      },
    {
        "name" : "Sofia"
      }
    ],
```

```
    "copies" : 0
  }
}
```

可以看到，在上面的代码中，数据并非是扁平的；它包含了数组和嵌套对象。如果要运用目前为止学到的知识创建映射，我们不得不将数据变为扁平。然而，Elasticsearch允许文档中存在一定程度的结构，可以创建能够处理上述示例的映射。

4.2.2　对象

上述示例显示了结构化的JSON文档。可以看到，示例文档的根对象是book，具备一些额外、简单的属性，比如englishTitle。它们将以正常字段的形式索引。此外，还有characters数组类型，这一点将在接下来的段落中讨论。现在，让我们关注author对象，可以看到，它还嵌套了另一个有两个属性firstName和lastName的对象name。

4.2.3　数组

我们已经使用过数组类型的数据，但未详细讨论。默认情况下，在Lucene中的所有字段都是多值的，因此在Elasticsearch中也是一样，这意味着它们可以存储多个值。为了索引这些字段，我们使用JSON数组类型，嵌套在中括号[]中。上述示例里，我们对book中的characters使用了数组类型。

4.2.4　映射

为索引数组，只需要在数组名称中指定字段的属性。因此，在我们的例子中，可添加以下映射来索引characters的数据：

```
"characters" : {
  "properties" : {
    "name" : {"type" : "string", "store" : "yes"}
  }
}
```

没什么特殊的，仅仅在数组名称（在例子中为characters）中嵌套了properties节点，并定义字段。由于前面的映射，我们在索引中获得多值字段characters.name。

同样，对于对象author，使用数据中同样的名称，除了properties部分，我们还添加了type属性并设置值为object，告知Elasticsearch应期待一个对象类型。在对象author中嵌套了对象name，因此author字段的映射如下所示：

```
"author" : {
  "type" : "object",
```

```
    "properties" : {
      "name" : {
        "type" : "object",
        "properties" : {
          "firstName" : {"type" : "string", "index" : "analyzed"},
          "lastName" : {"type" : "string", "index" : "analyzed"}
        }
      }
    }
  }
```

firstName和lastName字段在索引中体现为author.name.firstName和author.name.lastName。

其余字段为简单的核心类型，2.2节已经讨论过，这里不再赘述。

最终映射

所以，我们的映射文件structured_mapping.json的最终映射如下所示：

```
{
  "book" : {
    "properties" : {
      "author" : {
        "type" : "object",
        "properties" : {
          "name" : {
            "type" : "object",
            "properties" : {、
              "firstName" : {"type" : "string", "store": "yes"},
              "lastName" : {"type" : "string", "store": "yes"}
            }
          }
        }
      },
      "isbn" : {"type" : "string", "store": "yes"},
      "englishTitle" : {"type" : "string", "store": "yes"},
      "year" : {"type" : "integer", "store": "yes"},
      "characters" : {
        "properties" : {
          "name" : {"type" : "string", "store": "yes"}
        }
      },
      "copies" : {"type" : "integer", "store": "yes"}
    }
  }
}
```

可以看到，所有字段中store的属性值均设置为yes。这只是为了向你展示所有字段都可以正常索引。

4.2.5 向Elasticsearch发送映射

现在，映射已经完成，我们可测试确认其是否有效。这次将使用一种稍微不同的技术来创建索引和映射。首先，使用以下命令行创建library索引：

```
curl -XPUT 'localhost:9200/library'
```

接着，使用以下命令行，将映射发送至book类型：

```
curl -XPUT 'localhost:9200/library/book/_mapping' -d @structured_mapping.
json
```

现在，可使用以下命令行索引我们的示例数据：

```
curl -XPOST 'localhost:9200/library/book/1' -d @structured_data.json
```

4.2.6 动态还是非动态

我们知道，Elasticsearch是无模式的，这意味着不必创建前面的映射就可索引数据。Elasticsearch的动态行为默认是打开的，但可能想在索引的某些部分把它关掉。为此，可为指定字段增加属性dynamic，将值设置为false，该属性应该设置在与非动态对象的type属性相同的级别上。举例来说，如果我们希望对象author和name为非动态，应该将映射文件的相关部分修改成类似下面这样：

```
"author" : {
  "type" : "object",
  "dynamic" : false,
  "properties" : {
    "name" : {
      "type" : "object",
      "dynamic" : false,
      "properties" : {
        "firstName" : {"type" : "string", "index" : "analyzed"},
        "lastName" : {"type" : "string", "index" : "analyzed"}
      }
    }
  }
}
```

应记住，为此类对象增加新字段时，应更新映射。

> 你也可以在elasticsearch.yml配置文件中添加index.mapper.dynamic属性，将值设置为false，关掉动态映射功能。

4.3　使用嵌套对象

某些情况下嵌套对象可以很方便。基本上，通过使用嵌套对象，Elasticsearch允许我们连接一个主文档和多个附属文档。主文档及嵌套文档一同被索引，放置于索引的同一段上（实际在同一块上），确保为该数据结构获取最佳性能。更改文档也是一样的，除非使用更新API，你需要同时索引父文档和其他所有嵌套文档。

> 如果想阅读更多Lucene中嵌套对象的工作方式，可参考迈克·麦坎德利斯（Mike McCandless）的博客，链接如下：http://blog.mikemccandless.com/2012/01/searching-relational-content-with.html。

现在，我们来看示例。假设有一个服装店，需要存储每件T恤的尺寸和颜色，那么，标准的、非嵌套映射将类似于以下的代码（存储于cloth.json中）：

```
{
  "cloth" : {
    "properties" : {
      "name" : {"type" : "string"},
      "size" : {"type" : "string", "index" : "not_analyzed"},
      "color" : {"type" : "string", "index" : "not_analyzed"}
    }
  }
}
```

假设仅有一件XXL的红色T恤和一件XL的黑色T恤，示例文档将类似于以下代码：

```
{
  "name" : "Test shirt",
  "size" : [ "XXL", "XL" ],
  "color" : [ "red", "black" ]
}
```

然而，这个数据结构有个问题：假使客户要在商店搜索XXL黑色T恤，会产生什么结果？运行以下查询来检查（假设我们已使用映射创建索引并在其中建立了示例文档）：

```
curl -XGET 'localhost:9200/shop/cloth/_search?pretty=true' -d '{
  "query" : {
   "bool" : {
    "must" : [
     {
      "term" : { "size" : "XXL" }
     },
     {
      "term" : { "color" : "black" }
     }
    ]
   }
  }
}'
```

我们不应该得到结果，对吧？但事实上，Elasticsearch返回了如下文档：

```
{
  (...)
  "hits" : {
    "total" : 1,
    "max_score" : 0.4339554,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "cloth",
      "_id" : "1",
      "_score" : 0.4339554,
      "_source" : { "name" : "Test shirt",
                    "size" : [ "XXL", "XL" ],
                    "color" : [ "red", "black" ]}
    } ]
  }
}
```

这是因为，经过比对文档，在size字段和color字段上有我们需要的值。当然，这不是我们想要的。因此，修改映射，使用嵌套对象来分离color和size。最终的映射看起来如下所示（我们把这些映射存到cloth_nested.json文件中）：

4

```
{
  "cloth" : {
    "properties" : {
      "name" : {"type" : "string", "index" : "analyzed"},
      "variation" : {
        "type" : "nested",
        "properties" : {
          "size" : {"type" : "string", "index" : "not_analyzed"},
          "color" : {"type" : "string", "index" : "not_analyzed"}
        }
      }
    }
  }
}
```

可以看到，我们在cloth类型中引入了新对象variation，它是嵌套的（type属性设置为nested），表示想为嵌套文档建立索引。现在修改文档，添加variation对象，其中有两个属性：size和color。示例产品将如下所示：

```
{
  "name" : "Test shirt",
  "variation" : [
    { "size" : "XXL", "color" : "red" },
    { "size" : "XL", "color" : "black" }
  ]
}
```

组织文档结构，以便每个尺寸及其匹配颜色成为一个独立文档。然而，如果执行之前的查询，

将无任何文档返回。这是因为，对于嵌套文件，需要使用专门的查询。因此，查询如下（当然，我们已经再次创建了索引和类型）：

```
curl -XGET 'localhost:9200/shop/cloth/_search?pretty=true' -d '{
  "query" : {
    "nested" : {
      "path" : "variation",
      "query" : {
        "bool" : {
          "must" : [
            { "term" : { "variation.size" : "XXL" } },
            { "term" : { "variation.color" : "black" } }
          ]
        }
      }
    }
  }
}'
```

现在，上述查询将无法返回索引中的文档，因为无法找到尺寸XXL且颜色为黑色的嵌套文档。这里简单讨论一下我们的查询，可以看到，我们使用nested查询来查询嵌套文档。path属性指定了嵌套对象的名称（可以使用多个名称）。nested类型包括了一个标准查询部分。应注意的是，在嵌套对象中为字段名称指定完整的路径，在多级嵌套中很方便操作（这也是可能的）。

> 如果你想在嵌套对象的基础上过滤数据，可使用嵌套过滤器，它具备与嵌套查询相同的功能。更多相关信息，请参阅3.5节。

评分与嵌套查询

在查询过程中处理嵌套文档时，有一个附加属性。除path属性外，还有个score_mode属性，它允许我们定义如何从嵌套查询中计算得分。在Elasticsearch中可将此属性设置为如下值。

- avg：这是默认值。使用这个值时，Elasticsearch可在指定的嵌套查询中计算出平均值。该平均值包含在主查询的得分中。
- total：score_mode属性设置为此值时，Elasticsearch可对每个嵌套查询的得分求和。该值包含在主查询的得分中。
- max：score_mode属性设置为此值时，Elasticsearch可得出嵌套查询的最高得分。该值包含在主查询的得分中。
- none：score_mode属性设置为此值时，Elasticsearch不计算嵌套查询的得分。

4.4 使用父子关系

上一节已讨论了索引嵌套文档及其父文档的能力。然而，即使嵌套文档在索引中是作为独立文档检索的，除非使用更新API，否则还是无法更改单个嵌套文档。而在Elasticsearch中，我们可利用父子关系操作。请看以下内容。

4.4.1 索引结构和数据索引

在此，参考之前讨论嵌套文档时使用的示例：假想的服装店。然而我们希望的是：在每次变更后，无需索引整个文档即可更新尺寸和颜色。

1. 父文档映射

在父文档中，name是我们需要的唯一字段。因此，在shop索引中创建cloth类型，执行如下命令：

```
curl -XPOST 'localhost:9200/shop'
curl -XPUT 'localhost:9200/shop/cloth/_mapping' -d '{
  "cloth" : {
    "properties" : {
      "name" : {"type" : "string"}
    }
  }
}'
```

2. 子文档映射

为创建子文档映射，要在_parent属性中添加父类型的名称，在我们的示例中为cloth。因此，创建类型variation的命令行将如下所示：

```
curl -XPUT 'localhost:9200/shop/variation/_mapping' -d '{
  "variation" : {
    "_parent" : { "type" : "cloth" },
    "properties" : {
      "size" : {"type" : "string", "index" : "not_analyzed"},
      "color" : {"type" : "string", "index" : "not_analyzed"}
    }
  }
}'
```

我们无需指定连接父子文档的字段，因为默认情况下Elasticsearch会使用唯一标识符。如前面相关章节所述，唯一标识符以默认的形式存在于索引中。

3. 父文档

现在，我们来索引父文档。操作很简单，只要执行索引命令，示例如下：

```
curl -XPOST 'localhost:9200/shop/cloth/1' -d '{
  "name" : "Test shirt"
}'
```

上述命令中，我们指定的文档标识符为1。

4. 子文档

为索引子文档，需要使用parent参数提供父文档的相关信息，将该参数设置为父文档的标识符。所以，为索引父文档中的两个子文档，执行下面的命令：

```
curl -XPOST 'localhost:9200/shop/variation/1000?parent=1' -d '{
  "color" : "red",
  "size" : "XXL"
}'
```

同样，执行如下命令行索引第二个子文档：

```
curl -XPOST 'localhost:9200/shop/variation/1001?parent=1' -d '{
  "color" : "black",
  "size" : "XL"
}'
```

这样，我们索引了两个附加文档，它们是新类型，但是我们已为其指定标识符为1的父文档。

4.4.2　查询

我们已经索引了数据，现在需要恰当的查询来匹配拥有子文档数据的文档。当然，也可针对子文档来执行查询并检测其父文档是否存在。然而要注意的是，针对父文档执行查询时，子文档将无法返回，反之亦然。

1. 查询子文档中的数据

如果要寻找XXL号的红色衣服，可以运行如下命令行：

```
curl -XGET 'localhost:9200/shop/_search?pretty' -d '{
  "query" : {
    "has_child" : {
      "type" : "variation",
      "query" : {
        "bool" : {
          "must" : [
            { "term" : { "size" : "XXL" } },
            { "term" : { "color" : "red" } }
          ]
        }
      }
    }
  }
}'
```

查询很简单。类型has_child告知Elasticsearch我们想在子文档中搜索。为了指定感兴趣的子类型，指定type属性为子类型的名称。然后用一个标准的bool查询（已经讨论过），查询的结果仅包含父文档，示例如下：

```
{
  (...)
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "cloth",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "name" : "Test shirt" }
    } ]
  }
}
```

● top_children查询

除has_child查询之外，Elasticsearch还公开了top_children查询，它查询子文档但返回父文档。此查询可针对特定数量的子文档，示例如下：

```
{
  "query" : {
    "top_children" : {
      "type" : "variation",
      "query" : {
        "term" : { "size" : "XXL" }
      },
      "score" : "max",
      "factor" : 10,
      "incremental_factor" : 2
    }
  }
}
```

上述查询首先在100个子文档中运行（factor乘以size的默认参数10）。如果找到10个父文档（因为默认size的参数值为10），这些文档将返回并结束查询。然而，如果返回的父文档数量较少，且尚有子文档未经查询，那么另外20个子文档将被查询（incremental_factor参数乘以size），直到找到规定数量的父文档或者所有子文档查询结束为止。

top_children查询通过使用score参数指定得分的计算方式，可能的参数值包括：max（所有子查询得分的最大值）、sum（所有子查询得分的总和）或avg（所有子查询得分的平均值）。

2. 查询父文档中的数据

如果想要返回与父文档中指定数据匹配的子文档，可使用类似于has_child的查询：has_parent。然而，我们用父文档类型的值指定parent_type属性，而不是type属性。这么

这个查询将返回索引的子文档，而不是父文档：

```
curl -XGET 'localhost:9200/shop/_search?pretty' -d '{
  "query" : {
    "has_parent" : {
      "parent_type" : "cloth",
      "query" : {
        "term" : { "name" : "test" }
      }
    }
  }
}'
```

Elasticsearch的响应如下所示：

```
{
  (...)
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "variation",
      "_id" : "1000",
      "_score" : 1.0, "_source" : {"color" : "red","size" : "XXL"}
    }, {
      "_index" : "shop",
      "_type" : "variation",
      "_id" : "1001",
      "_score" : 1.0, "_source" : {"color" : "black","size" : "XL"}
    } ]
  }
}
```

4.4.3　父子关系和过滤

如果想要将父子查询作为过滤器使用，可以用过滤器`has_child`和`has_parent`，它们具备了与`has_child`和`has_parent`查询相同的功能。实际上，Elasticsearch将那些过滤器封装为常数得分查询，使其可作为查询使用。

4.4.4　性能考虑

使用Elasticsearch父子的功能时，必须注意它的性能影响。需要记住的第一件事是父子文档需要存储在相同的分片中，查询才能够工作。如果单一父文档有大量的子文档，可能导致分片上的文档数量不平均。因此，其中的一个节点的性能会降低，造成整个查询速度变慢。另外，请记住，比起查询无任何关联的文档，父子查询的速度较慢。

第二个非常重要的事情是，执行has_child等查询时，Elasticsearch需要预加载并缓存文档标识符。这些标识符将存储在内存中，必须确保Elasticsearch有足够的内存。否则，你将得到OutOfMemory异常，节点或整个集群将无法运作。

最后，我们提到过，首次查询将花一定时间预加载和缓存文档标识符。为了提升首次查询父子关系文档的性能，可以使用预热API。关于如何在Elasticsearch中添加预热查询，请参考8.5节。

4.5 使用更新 API 修改索引结构

前面的章节讨论过如何创建索引映射和索引数据。如果你已经创建了映射和索引数据，想要修改索引的结构，应该怎么办？在某种程度上这是可行的。例如，默认情况下索引一个带新字段的文档，Elasticsearch会将该字段增加到索引结构中。现在看看如何手动修改索引结构。

4.5.1 映射

假设我们的users索引有以下映射，存储于user.json文件中：


```
{
  "user" : {
    "properties" : {
      "name" : {"type" : "string"}
    }
  }
}
```

可以看到，它很简单，只有一个属性保存用户名。现在，让我们创建一个名为users的索引，并使用上面的映射创建自己的类型。为此，运行以下命令：

```
curl -XPOST 'localhost:9200/users'
curl -XPUT 'localhost:9200/users/user/_mapping' -d @user.json
```

如果一切正常，我们的索引和类型便创建好了。现在，添加一个新字段到映射中去。

4.5.2 添加一个新字段

为了说明如何为映射添加新字段，我们假设要为每个存储的用户添加一个电话号码。为此，需要将HTTP PUT命令发送到带有合适主体的/index_name/type_name/_mapping REST端点，该主体中包含我们的新字段。例如，为添加phone字段，执行以下命令：

```
curl -XPUT 'http://localhost:9200/users/user/_mapping' -d '{
  "user" : {
    "properties" : {
      "phone" : {"type" : "string",
```

```
                    "store" : "yes",
                    "index" : "not_analyzed"}
      }
    }
}'
```

同样，如果一切正常，新字段便添加到我们的索引结构中去了。为了确保一切正常，可以运行HTTP GET请求到_mapping端点；Elasticsearch将返回适当的映射。在索引users中获得user类型映射的示例命令如下所示：

```
curl -XGET 'localhost:9200/users/user/_mapping?pretty'
```

> 在现有类型中添加新字段后，需要再次对所有文档进行索引，因为Elasticsearch不会自动更新。这很关键，可以使用初始数据源或从_source字段中获得初始数据并再次索引。

4.5.3　修改字段

现在，我们的索引结构包含两个字段：name和phone。我们索引了一些数据，但之后又决定搜索phone字段，并希望更改index属性，从not_analyzed改为analyzed，为此，执行以下命令：

```
curl -XPUT 'http://localhost:9200/users/user/_mapping' -d '{
  "user" : {
    "properties" : {
      "phone" : {"type" : "string",
                 "store" : "yes",
                 "index" : "analyzed"}
    }
  }
}'
```

执行上面的命令行后，Elasticsearch返回以下输出：

```
{"error":"MergeMappingException[Merge failed with failures {[mapper
[phone] has different index values, mapper [phone] has different 'norms.
enabled' values, mapper [phone] has different tokenize values, mapper
[phone] has different index_analyzer]}]","status":400}
```

这是因为无法将not_analyzed字段更改为analyzed。不仅如此，在大部分情况下字段映射是无法更新的。这是好事，因为如果我们可以更改这样的设置，会让Elasticsearch和Lucene混乱。假设已经有很多文档带有设置为not_analyzed的phone字段，将这些设置更改为analyzed，Elasticsearch将无法更改已索引的文档，但已经分析的查询将以不同的逻辑处理，那么我们就无法正确查找数据。

我们在此将提及一些操作，举例说明哪些是禁止的，哪些是允许的。如下修改是安全的：

- 增加新的类型定义；
- 增加新的字段；
- 增加新的分析器。

而以下修改是不允许或是无法实现的：

- 更改字段类型（如将文本改为数字）；
- 更改“存储到”字段为不存储，反之亦然；
- 更改索引属性的值；
- 更改已索引文档的分析器。

注意一点，上述允许和不允许的操作没有涵盖更新API的全部可能性，你必须实际操作以验证更新是否可行。

> 如果你想忽略冲突并设置新映射，可设置ignore_conflicts的参数值为true，Elasticsearch将会重写映射。带有额外参数的命令行如下：
>
> ```
> curl -XPUT 'http://localhost:9200/users/user/_mapping?
> ignore_conflicts=true' -d '...'
> ```


4.6 小结

这一章讲述了如何使用Elasticsearch索引树型结构，索引非扁平数据，以及修改已创建的索引结构。最后，介绍了如何使用嵌套文档及Elasticsearch中的父子功能来处理关系型数据。

下一章的重点是更高效地搜索，我们将看到Apache Lucene得分的工作方式及其重要性；学习使用Elasticsearch的函数得分查询，用函数以及提供的脚本功能来调整不同文档的重要性，使用不同的语言来搜索，并讨论索引时加权什么时候有意义。下一章将使用同义词匹配具有相同含义的单词，介绍检测文档被查询到的原因。最后，使用加权来影响查询，并解释Elasticsearch的得分计算。

第5章 更好的搜索

上一章介绍了Elasticsearch如何对非扁平数据建立索引，如何索引树状结构和面向对象结构的数据，以及如何修改已创建索引的结构。最后，还学习了如何使用嵌套文档和父子功能来处理文档之间的关系。本章主要内容如下：

- Apache Lucene评分；
- 使用Elasticsearch的脚本功能；
- 对不同语言的数据索引和搜索；
- 使用不同查询来影响返回文档的得分；
- 使用索引时加权；
- 具有相同意思的词；
- 检查为什么特定文档被返回；
- 检查得分计算细节。

5.1 Apache Lucene 评分简介

当谈到查询及其相关性，我们不能忽略得分以及它从哪里来。但得分是什么？得分是描述文档与查询相关度的一个参数。本节将讨论Apache Lucene的默认评分机制：TF/IDF算法，看看它如何影响返回的文档。

> TF/IDF算法不是Elasticsearch公开的唯一可用的算法。有关可用模型的更多信息，请参阅2.2.3节，或我们的书*Mastering ElasticSearch*，Packt出版。

5.1.1 当文档被匹配时

Lucene返回文档时，意味着文档与我们发送的查询匹配，并且对该文档已给出一个分数。得分越高，从搜索引擎的角度来看文档越相关。然而，两个不同的查询将对同一文档计算出不同的分数。正因为如此，在查询之间比较分数通常没什么意义。我们回到评分这个话题。计算文档的

评分属性时，考虑以下因素。

- 文档加权：对文档建立索引时，对文档的加权值。
- 字段加权：查询和索引时，对字段的加权值。
- 协调：基于文档词条数的协调因子。对包含更多查询词条的文档，它提供更大的值。
- 逆文档频率：基于词条的因子，它告诉评分公式，给定词条出现的频率有多低。逆文档频率越高，词条越罕见。
- 长度规范：基于字段的规范化因子，它基于给定字段包含的词条数目。字段越长，该因子给的加权值越小。这基本上意味着更短的文档更受分数的青睐。
- 词频：基于词条的因子，描述给定词条在文档中出现的次数，词频越高，文档的得分越高。
- 查询规范：基于查询的规范化因子，由每个查询词条比重的平方之和计算而成。查询规范用于查询之间的得分比较，但这并不一定很容易，有时甚至做不到。

5.1.2 默认评分公式

TF/IDF算法的实用计算公式如下：

$$score(q,d) = coord(q,d) * queryNorm(q) * \sum_{tinq}(tf(tind) * idf(t)^2 * boost(t) * norm(t,d))$$

为了调整查询相关性，你不需要记住这个等式的细节，但至少要知道它是如何工作的。我们可以看到，文档的评分因子是查询q和文档d的一个函数。还有两个不直接依赖于查询词条的因子，coord和queryNorm。公式中这两个元素跟查询中的每个词计算而得的总和相乘。另一方面，该总和由给定词的词频、逆文档频率、词条加权和规范相乘而来，其中的规范就是我们前面讨论过的长度规范。

> 注意前面的公式是实用性的，你可以在Lucene Javadocs中查看更多概念公式的信息，网址是：http://lucene.apache.org/core/4_7_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html。

上述规则的好处是，你不需要记住全部内容。应该知道的是影响文档评分的因素。下面是一些派生自上述等式的规则：

- 匹配的词条越罕见，文档的得分越高；
- 文档的字段越小，文档的得分越高；
- 字段的加权越高，文档的得分越高；
- 我们可以看到，文档匹配的查询词条数目越高、字段越少（意味着索引的词条越少），Lucene给文档的分数越高。同时，罕见词条比常见词条更受评分的青睐。

5.1.3　相关性的意义

在大多数情况下，我们希望得到最匹配的文档，但最相关的不一定是最匹配的。一些用例定义了非常严格的规则，规定了某些文档应该在结果列表中排位靠前。例如，文档除了被TF/IDF相似度模型完美匹配外，有客户付钱让他们的文档出现在结果中更靠前的位置。基于客户计划，我们想给这样的文档更大的重要性，把付费最高的用户的文档放到搜索结果的最顶部。当然，这就不属于TF/IDF相关性了。

这是一个非常简单的例子，但Elasticsearch查询可以非常复杂。5.4节将讨论。

在进行搜索相关性方面的工作时，你应该永远记住，这不是一次性的过程。随着时间的推移，你的数据将改变，查询也需要相应调整。在大多数情况下，优化查询相关性是持续性的工作，要根据业务规则、需求以及用户行为方式等调整。有一点非常重要：记住这不是设置之后就可以抛诸脑后的一次性过程。

5.2　Elasticsearch 的脚本功能

Elasticsearch有几个可以使用脚本的功能。你已经看过一些例子，如更新文件、过滤和搜索。这似乎有点高级，我们还是来看看Elasticsearch提供的可能性，因为在一些用例中，脚本是非常有价值的。Elasticsearch使用脚本执行的任何请求中，我们会注意到以下相似的属性。

- `script`：此属性包含实际的脚本代码。
- `lang`：这个属性定义了提供脚本语言信息的字段。如果省略，Elasticsearch假定为`mvel`。
- `params`：此对象包含参数及其值。每个定义的参数可以通过指定参数名称在脚本中使用。通过使用参数，我们可以编写更干净的代码。由于可以缓存，使用参数的脚本比嵌入常数的代码执行得更快。

5.2.1　脚本执行过程中可用的对象

在不同的操作过程中，Elasticsearch允许在脚本中使用不同的对象。为开发符合我们用例的脚本，应该熟悉这些对象。

比如，在搜索过程中，下列对象是可用的。

- `_doc`（也可以用`doc`）：这是个`org.elasticsearch.search.lookup.DocLookup`对象的实例。通过它可以访问当前找到的文档，附带计算的得分和字段的值。
- `_source`：这是个`org.elasticsearch. search.lookup.SourceLookup`对象的实例，通过它可以访问当前文档的`source`，以及定义在`source`中的值。

- _fields：这是个org.elasticsearch.search.lookup.FieldsLookup对象的实例，通过它可以访问文档的所有字段。

另一方面，在文档更新过程中，Elasticsearch只通过_source属性公开了ctx对象，通过它可以访问当前文档。

我们之前看到过，在文档字段和字段值的上下文中提到了几种方法。现在让我们通过下面的例子，看看如何获取title字段的值。在括号中，你可以看到Elasticsearch从library索引中为我们的一个示例文档返回的值：

- _doc.title.value(crime)
- _source.title (Crime and Punishment)
- _fields.title.value (null)

有点疑惑，不是吗？在索引期间，一个字段值作为_source文档的一部分被发送到Elasticsearch。Elasticsearch可以存储此信息，而且默认的就是存储。此外，文档被解析，每个被标记成stored的字段可能都存储在索引中（也就是说，store属性设置为true；否则，默认情况下，字段不存储）。最后，字段值可以配置成indexed。这意味着分析该字段值，划分为标记，并放置在索引中。综上所述，一个字段可能以如下方式存储在索引中：

- 作为_source文档的一部分；
- 一个存储并未经解析的值；
- 一个解析成若干标记的值。

5

在脚本中，除了更新操作，我们可以访问所有这些形式。你可能疑惑应该使用哪个版本。好吧，如果我们要访问处理过的形式，答案很简单，_doc。那_source和_fields呢？在大多数情况下，_source是一个不错的选择，比起从索引中读取原始字段值来说，它通常很快并且磁盘操作较少。

5.2.2 MVEL

Elasticsearch可以在脚本中使用几种语言。如果没有明确说明，默认使用MVEL（MVFLEX Expression Language，MVFLEX表达式语言）。MVEL快速、易于使用和嵌入，是在很多开源项目中使用的一个简单但功能强大的表达式语言。它允许我们使用Java对象，自动映射属性到getter/setter方法、简单类型转换、集合映射、数组和关联数组映射等。更多关于MVEL的信息，请参阅http://mvel.codehaus.org/Language+Guide+for+2.0。

5.2.3 使用其他语言

脚本中使用MVEL是一个简单而充分的解决方案，但你也可以选择JavaScript、Python或者

Groovy。使用其他语言之前，必须安装相应的插件，可以在8.7节中阅读更多关于插件的内容。现在，我们只需从Elasticsearch目录中执行以下命令：

```
bin/plugin -install elasticsearch/elasticsearch-lang-
  javascript/2.0.0.RC1
```

上述命令将安装一个插件，使我们能够使用JavaScript。我们在请求中的唯一修改是添加所使用脚本语言的额外信息，当然，还有修改脚本本身，让它在新的语言中是正确的。看看下面的示例：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc.tags.values.length > 0 ? doc.tags.values[0]
        :'\u19999';",
      "lang" : "javascript",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

可以看到，我们使用JavaScript脚本，而不是默认MVEL。`lang`参数通知Elasticsearch我们正在使用的语言。

5.2.4　使用自定义脚本库

脚本通常都很小，可以很方便地放到请求中。但有时随着应用程序的增长，你想给开发者一些可以在他们的模块中重复使用的东西。如果是大型和复杂的脚本，一般最好将它们放在文件中，并仅在API请求中引用。要做的第一件事情是把脚本用适当的名称放在适当的地方。我们的小脚本应该放在Elasticsearch的config/scripts目录中。把我们的示例脚本文件命名为text_sort.js。请注意，文件的扩展名应该指示用于脚本的语言。在本例中，使用JavaScript。本示例文件的内容很简单，如下所示：

```
doc.tags.values.length > 0 ? doc.tags.values[0] :'\u19999';
```

一个使用上述脚本的查询如下所示：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
```

```
        "script" : "text_sort",
        "type" : "string",
        "order" : "asc"
      }
    }
  }
```

你可以看到，现在可以使用text_sort作为脚本的名称。此外，可以省略脚本语言，Elasticsearch会从文件扩展名判断它。

使用本地代码

如果脚本太慢，或者你不喜欢脚本语言，Elasticsearch允许你使用Java类，而不是脚本。

(1) 工厂实现类

需要实现至少两个类来创建新的本地脚本。第一个是我们脚本的工厂。现在，先关注它。下面的代码示例说明了我们的脚本工厂：

```
package pl.solr.elasticsearch.examples.scripts;

import java.util.Map;
import org.elasticsearch.common.Nullable;
import org.elasticsearch.script.ExecutableScript;
import org.elasticsearch.script.NativeScriptFactory;

public class HashCodeSortNativeScriptFactory implements
  NativeScriptFactory {

  @Override
  public ExecutableScript newScript(@Nullable Map <string, Object>
    params) {
    return new HashCodeSortScript(params);
  }
}
```

重要部分是高亮的代码片段。这个类必须实现org.elasticsearch.script.NativeScriptFactory类。该接口强制我们实现newScript()方法。它接收定义在API请求中的参数，返回脚本的一个实例。

(2) 实现本地脚本

现在让我们看看脚本的实现。想法很简单：我们的脚本将用于排序。文档将按照选择字段的hashCode()值来排序，没有定义该字段的文档将是第一个。我们知道此逻辑没有太多意义，但它很简单，是个好例子。本地脚本源代码如下所示：

```
package pl.solr.elasticsearch.examples.scripts;

import java.util.Map;
```

```
import org.elasticsearch.script.AbstractSearchScript;

public class HashCodeSortScript    extends AbstractSearchScript {
  private String field = "name";

  public HashCodeSortScript(Map <string, Object> params) {
    if (params != null && params.containsKey("field")) {
      this.field = params.get("field").toString();
    }
  }

  @Override
  public Object run() {
    Object value = source().get(field);
    if (value != null) {
      return value.hashCode();
    }
    return 0;
  }
}
```

首先，我们的类从`org.elasticsearch.script.AbstractSearchScript`类继承并实现`run()`方法。该方法从当前文档中得到适当的值，并且根据我们的奇怪逻辑来处理后，返回一个结果。你可能注意到了`source()`，没错，它和我们在非本地脚本中碰到的`_source`参数完全一样。`doc()`和`fields()`方法同样是可用的，与之前描述的逻辑一样。

值得一看的是我们使用这些参数的方式。假设用户会填充`field`参数，告诉我们文档的哪个字段将被用来操纵。我们还提供了此参数的默认值。

(3) 安装脚本

现在来安装本地脚本。在把已编译的类封装成JAR归档后，应该把它放在Elasticsearch的lib目录，这使我们的代码对类加载器可见。我们应该做的是注册脚本，可以通过Setting API调用来实现或在elasticsearch.yml配置文件添加一行。这里选择使用elasticsearch.yml脚本，将下面这行添加到上述配置文件：

```
script.native.native_sort.type:
  pl.solr.elasticsearch.examples.scripts.
    HashCodeSortNativeScriptFactory
```

注意`native_sort`片段。这是在请求期间使用的脚本名称，它将会被传递给`script`参数。此属性的值是工厂实现的完整类名，它将用来初始化脚本。最后，需要重新启动Elasticsearch。

(4) 执行脚本

已经重新启动Elasticsearch，所以，可以开始使用本地脚本发送查询。例如，发送一个查询，使用之前的`library`索引中的数据。示例查询如下所示：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "native_sort",
      "params" : {
        "field" : "otitle"
      },
      "lang" : "native",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

注意查询的params部分，在这个请求中，要对otitle字段排序。提供native_sort作为脚本名字、native作为脚本语言，这是必需的。如果一切顺利，应该看到结果按自定义排序逻辑排序。在Elasticsearch的响应中，我们会看到，没有otitle字段的文档将出现在结果列表的最上面，它们的sort值为0。

5.3 搜索不同语言的内容

此前，在讨论语言的分析时，我们主要谈理论，还没看到一个关于语言分析、处理数据可能包含的多语言例子。现在，我们将讨论如何处理多语言的数据。

5.3.1 区分处理不同语言

你已经知道，Elasticsearch能为数据提供不同的分析器，可以让数据以空白符分割、小写，等等。这通常可以处理不同语言的数据：应该可以把基于空白符的分词器用在英语、德语和波兰语（但不适用于中文）。然而，如果你只想发送单词cat到Elasticsearch，找包含cat和cats的文档，应该怎么办？这是语言分析起作用的地方，使用不同语言的词干提取算法，分析单词，回退到词根状态。

现在最糟糕的部分是，不能用一个通用的词干提取算法来处理世界上所有的语言；要选择一个合适的语言。以下章节将帮助你理解语言分析过程的某些部分。

5.3.2 多语言处理

Elasticsearch中有几种处理多语言的方法，它们都有利有弊。我们不会讨论每一个，为了让你有所了解，列出如下方法：

- 把不同语言的文档存储成不同的类型；
- 把不同语言的文档存储到单独的索引中；
- 对单一文档的字段存储多个版本，每个版本包含不同的语言。

不过，我们会将重心放在一个能够将多语言文档存储在单个索引的方法。将注意力集中在这个问题上：文档只有一个类型，但它们可能来自世界各地，因此可以有多种语言。同时，我们希望用户在不同语言上使用所有的分析功能，如词干提取和停止词，而不仅是英语。

> 请注意，不同语言上的词干提取算法是不一样的：不管是分析性能还是词条结果。例如，英语词干分析器很好，但在欧洲语言（比如德语）上执行时，可能会有问题。

5.3.3　检测文档的语言

如果你不知道文档或查询的语言（大多时候是这样），可以使用语言检测软件，在一定程度上可以检测文档或查询的语言。如果使用Java，可以使用几个可用的语言检测库之一。比如下面这些：

- Apache Tika（http://tika.apache.org/）；
- Language detection（http://code.google.com/p/language-detection/）。

Language detection库声称支持53种语言并提供99%的准确度，可以说很多了。

应该记住，文本越长语言检测越准确。然而，由于查询的文本通常很短，你可能会在查询语言识别过程中遇到一定程度的错误。

5.3.4　示例文档

先介绍一个示例文档，如下所示：

```
{
  "title" : "First test document",
  "content" : "This is a test document",
  "lang" : "english"
}
```

可以看到，该文档很简单，包含如下3个字段。

- `title`：该字段存储文档的标题。
- `content`：该字段存储文档的实际内容。
- `lang`：该字段定义识别语言。

前两个字段从用户的文档创建而来，第三个字段是我们的假想用户在上传文档时选择的。

为了通知Elasticsearch选择哪个分析器，我们把`lang`字段映射到Elasticsearch的分析器之一（分析器的完整列表可以在这里找到：http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html）。如果用户输入了一个不支持的语言，就不设置`lang`字段，让Elasticsearch使用默认分析器。

5.3.5 映射文件

来看看为上述文档创建的映射，把它们存储在mappings.json文件中，如下所示：

```
{
  "mappings" : {
    "doc" : {
      "_analyzer" : {
        "path" : "lang"
      },
      "properties" : {
        "title" : {
          "type" : "string",
          "index" : "analyzed",
          "store" : "no",
          "fields" : {
            "default" : {
              "type" : "string",
              "index" : "analyzed",
              "store" : "no",
              "analyzer" : "simple"
            }
          }
        },
        "content" : {
          "type" : "string",
          "index" : "analyzed",
          "store" : "no",
          "fields" : {
            "default" : {
              "type" : "string",
              "index" : "analyzed",
              "store" : "no",
              "analyzer" : "simple"
            }
          }
        },
        "lang" : {
          "type" : "string",
          "index" : "not_analyzed",
          "store" : "yes"
        }
      }
```

```
    }
  }
}
```

上面的映射中，我们最感兴趣的是分析器定义，以及`title`和`description`字段（如果你还不熟悉映射，请参阅2.2节）。我们希望分析器基于`lang`字段，因此，要在`lang`字段添加一个与Elasticsearch知道的分析器的名字相同的值，可以是默认的，也可以是自定义的。

其次是拥有实际数据的两个字段的定义。你可以看到，我们使用多字段定义来索引`title`和`description`字段。多字段的第一个使用`lang`字段指定的分析器（因为没有指定分析器名字，所以使用全局定义的那个）来建立索引，如果查询时知道指定的语言，也将使用这个字段。多字段的第二项使用`simple`分析器，不知道查询语言时，使用这个字段。`simple`分析器只是个例子，你同样可以使用标准分析器或其他任意分析器。

为使用这个映射文件创建一个叫`docs`的简单索引，执行如下命令：

```
curl -XPUT 'localhost:9200/docs' -d @mappings.json
```

5.3.6　查询

现在看看如何查询数据。把查询分成下面两种情况。

1. 用识别语言查询

第一种情况是确定了查询语言。假设识别的语言是英语，我们知道英语与`english`分析器匹配。在这种情况下，查询如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "match" : {
      "content" : {
        "query" : "documents",
        "analyzer" : "english"
      }
    }
  }
}'
```

`analyzer`参数指明了我们想用的分析器。将该参数设置为识别语言所对应的分析器的名字。注意，我们正在寻找的词条是`documents`，而在文档中的词条是`document`。`english`分析器可以处理这个情况并找到文档。Elasticsearch返回的响应如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
```

```
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 0.19178301
    } ]
  }
}
```

2. 用未知语言查询

假设不知道用户查询使用的语言。此时，不能使用由lang字段指定的分析器，因为我们不想用一个特定于语言的分析器来分析查询。在这种情况下，用标准的简单分析器，发送查询到contents.default字段，而不是content字段。查询如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "match" : {
      "content.default" : {
        "query" : "documents",
        "analyzer" : "simple"
      }
    }
  }
}'
```

然而，这次没有得到任何结果，因为搜索单词的复数形式时，simple分析器不能处理它的单数形式。

3. 组合查询

为了对完美匹配默认分析器的文档额外加权，可以把上述两个查询用bool查询组合起来，如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "bool" : {
      "minimum_should_match" : 1,
      "should" : [
        {
          "match" : {
            "content" : {
              "query" : "documents",
              "analyzer" : "english"
            }
```

```
          }
        },
        {
          "match" : {
            "content.default" : {
              "query" : "documents",
              "analyzer" : "simple"
            }
          }
        }
      ]
    }
  }
}'
```

返回的文档至少必须匹配一个定义的查询。如果两者都匹配，文档将具有更高的分数值，并在结果中放置得更高。

前面的组合查询有一个额外的优势：如果我们的语言分析器找不到一个文档（例如，所用的分析跟在索引期间使用的不一样时），第二个查询有机会找到只用空白字符分词和小写形式的词条。

5.4　使用查询加权影响得分

上一章介绍了什么是得分以及Elasticsearch如何计算它。随着应用程序的增长，提高搜索质量的需求也进一步增大。我们把它叫做搜索体验。我们需要知道什么对用户更重要，关注用户如何使用搜索功能。这导致不同的结论，例如，有些文档比其他的更重要，或特定查询需强调一个字段而弱化其他字段。这就是可以用到加权的地方。

5.4.1　加权

*加权*是一个评分过程中额外使用的值。我们已经知道它适用于下列地方。

- `query`：这可以通知搜索引擎，给定查询是复杂查询的一部分，而且比其他部分更重要。
- `field`：有几个文档字段对用户非常重要。例如，以Bill搜索电子邮件,应该首先列出那些发送自Bill的邮件，紧随其后列出主题中含有Bill的邮件，然后是内容中提到Bill的邮件。

指定给查询或字段的加权值只是计算分数时的众多因素之一，我们都应意识到这一点。现在，看几个查询的例子。

5.4.2　为查询添加加权

假设索引有两个文档，第一个文档如下所示：

```
{
  "id" : 1,
  "to" : "John Smith",
  "from" : "David Jones",
  "subject" : "Top secret!"
}
```

第二个文档如下所示：

```
{
  "id" : 2,
  "to" : "David Jones",
  "from" : "John Smith",
  "subject" : "John, read this document"
}
```

数据很简单，但它应该很好地描述了我们的问题。现在，假设有以下查询：

```
{
  "query" : {
    "query_string" : {
       "query" : "john",
       "use_dis_max" : false
    }
  }
}
```

在这个例子中，Elasticsearch将为_all字段创建一个查询，并将查找包含所需文字的文档。通过把use_dis_max参数设为false，告诉Elasticsearch不希望使用disjunction查询（如果你不记得disjunction查询，请参阅3.3节中的最大分查询和字符串查询部分）。很容易猜到，我们的两条记录都将返回，标识符等于2的那个将第一个返回。这是由于John分别出现在from字段和subject字段。检查一下结果：

```
"hits" : {
  "total": 2,
  "max_score": 0.13561106,
  "hits": [{
    "_index": "messages",
    "_type": "email",
    "_id": "2",
    "_score": 0.13561106,"_source":
    {"id": 1,"to": "David Jones","from":
    "John Smith","subject": "John, read this document"}
  }, {
    "_index": "messages",
    "_type": "email",
    "_id": "1",
    "_score": 0.11506981,"_source":
    {"id": 2,"to": "John Smith","from":
    "David Jones","subject": "Top secret!"}
  } ]
}
```

一切都正常吗？技术上来说，是的。但我认为第二个文档应该出现在结果列表的第一位，因为搜索时，在许多情况下，最重要的因素是匹配人，而不是消息主题。你可能不同意，但这正好解释了为什么全文搜索的相关性是一个困难的课题：有时，很难判断在特定情况下哪个排序更好。我们能做什么？首先，重写查询，间接告知Elasticsearch应使用哪些字段搜索，如下所示：

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from", "to", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

这和上一个示例查询不完全一样。运行它，将得到同样的结果，但如果仔细观察，你会发现在得分上的差异。在上一个示例中，Elasticsearch只使用一个字段，`_all`。现在我们在3个字段上搜索。这意味着有些因素改变了，如字段长度等。不过，这在我们的例子中不是那么重要。在底层，Elasticsearch生成由3个查询组成的复杂查询：每个字段一个查询。当然，每个查询的贡献得分取决于这个字段上找到的词条数以及这个字段的长度。我们介绍一些字段之间的差异。将下面的查询同前面的查询相比较：

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from^5", "to^10", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

看看高亮显示的部分（^5和^10）。通过这种方式，可以告诉Elasticsearch给定字段的重要程度。我们看最重要的字段是`to`，其次是`from`。`subject`字段的`boost`为默认值，即`1.0`。永远记住，这个值只是各种因素之一。你可能想知道为什么选择5，而不是1000或1.23。嗯，这取决于我们想要达到的效果，有什么样的查询，更重要的是在索引中有什么样的数据。通常，当数据有意义的部分发生变化时，也许我们应该再次检查和调整相关性。最后，看一个类似的例子，但这次使用`bool`查询，如下所示：

```
{
  "query" : {
    "bool" : {
      "should" : [
        { "term" : { "from": { "value" : "john", "boost" : 5 }}},
        { "term" : { "to": { "value" : "john", "boost" : 10  }}},
        { "term" : { "subject": { "value" : "john" }}}
      ]
```

```
      }
    }
  }
```

5.4.3　修改得分

前面的示例演示了如何通过加权特定查询组件来影响结果列表。另一种方法是运行一个查询来影响匹配文档的得分。接下来几节将总结Elasticsearch提供的几种可能性。我们将在例子中使用第3章中用过的数据。

1. **constant_score**查询

constant_score查询允许我们对任何过滤器或查询明确设置一个被用作得分的值，它将通过加权参数赋给每个匹配文档。

乍看起来，此查询并不实际。但考虑到建立复杂查询的情况，这种查询允许我们设置多少个匹配查询的文档可以影响总分。看看下面的示例：

```
{
  "query" : {
    "constant_score" : {
      "query": {
        "query_string" : {
          "query" : "available:false author:heller"
        }
      }
    }
  }
}
```

在我们的数据中，有两个文档的available字段为false，其中一个在author字段上有值。但由于constant_score查询，Elasticsearch将在评分时忽略此信息，两个文档的得分都是1.0。

2. 加权查询

下一个与加权相关的查询是加权查询。它允许我们定义一个查询的额外部分，用于降低匹配文档的得分。下面的例子列出所有图书，但E.M.Remarque写的书得分将低10倍：

```
{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "available" : true
        }
      },
      "negative" : {
        "match" : {
          "author" : "remarque"
```

```
        }
      },
      "negative_boost" : 0.1
    }
  }
}
```

3. function_score查询

我们已经看过两个允许改变查询返回文档得分的例子。第三个例子，我们想谈谈function_score查询，它比之前的查询更复杂。这个查询在得分计算成本高昂时非常有用，因为它将计算过滤后文档的得分。

(1) 函数查询的结构

函数查询的结果很简单，看上去如下所示：

```
{
  "query" : {
    "function_score" : {
      "query" : { ... },
      "filter" : { ... },
      "functions" : [
       {
         "filter" : { ... },
         "FUNCTION" : { ... }
       }
     ],
      "boost_mode" : " ... ",
      "score_mode" : " ... ",
      "max_boost" : " ... ",
      "boost" : " ... "
    }
  }
}
```

一般来说，function_score查询中可以使用查询、过滤、函数和附加参数。每个函数可以有一个过滤器定义要应用的过滤结果。如果一个函数没有定义过滤器，它将应用到所有文档。

function_score查询背后的逻辑很简单。首先，函数匹配文档并基于score_mode参数计算得分。然后，文档的查询得分由函数计算所得分数结合而成，结合时基于boost_mode参数。

我们现在来讨论以下参数。

- boost_mode：boost_mode参数允许定义如何将函数查询所计算分数与查询分数结合起来。可以将它设置成下列值。
  - multiply：这是默认行为，查询得分将与函数计算所得分相乘。
  - replace：导致查询得分全部被忽略，文档得分等于函数计算所得分。

- sum：文档得分等于查询得分和函数得分的总和。
- avg：文档得分等于查询得分和函数得分的平均值。
- max：文档得分等于查询得分和函数得分的最大值。
- min：文档得分等于查询得分和函数得分的最小值。

❑ score_mode：该参数定义了函数计算所得分是如何结合在一起的。以下是该参数可以设置的值。

- multiply：这是默认行为，结果是查询得分乘以函数的得分。
- sum：把所定义的函数的得分相加。
- avg：函数得分等于所有匹配函数得分的平均值。
- first：把第一个拥有匹配文档过滤器的函数的得分返回。
- max：返回所有函数得分的最大值。
- min：返回所有函数得分的最小值。

要记住，可以通过使用function_score查询中的max_boost参数来限制最大计算得分。默认情况下，这个参数的值被设为Float.MAX_VALUE，意思是最大浮点值。

boost参数允许我们为文档设置一个查询范围的加权值。

我们还没谈到可以包含到查询的functions节点中的函数，下面是目前可用的函数。

❑ boost_factor函数：这个函数允许我们把文档分数乘以一个给定值。boost_factor参数的值不会被范式化，而是原原本本的值。下面是使用boost_factor参数的一个例子：

```
{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        { "boost_factor" : 20 }
      ]
    }
  }
}
```

❑ script_score函数：这个函数允许我们使用一个脚本来计算得分，用于函数返回的得分（然后将落到boost_mode参数定义的行为中去）。使用script_score函数的例子如下所示：

```
{
  "query" : {
```

5

```
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        {
          "script_score" : {
            "script" : "_score * _source.copies *
              parameter1",
            "params" : {
              "parameter1" : 12
            }
          }
        }
      ]
    }
  }
}
```

- random_score函数：使用这个函数，可以通过指定seed值来生成一个伪随机分数。为了模拟随机性，每次都应指定一个新的seed。一个使用此功能的例子如下所示：

```
{
  "query" : {
    "function_score" : {
     "query" : {
       "term" : {
         "available" : true
       }
      },
      "functions" : [
        {
          "random_score" : {
            "seed" : 12345
          }
        }
      ]
    }
  }
}
```

- decay函数：除了前面提到的评分函数以外，Elasticsearch包含了额外的decay函数。前述函数给出的分数随着距离变大而变低，该函数则不同。距离基于一个单值的数值型字段（比如日期、地理位置点或标准的数值型字段）计算而来。最容易想到的例子是基于与一个给定点的距离来加权文档。

假设有一个point字段，存储着位置信息，我们希望文档的得分受与用户所在位置的距离影响（比如，用户用手机设备发送请求），用户在52，21这个位置，我们可以发送如下查询：

```
{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        {
          "linear" : {
            "point" : {
              "origin" : "52, 21",
              "scale" : "1km",
              "offset" : 0,
              "decay" : 0.2
            }
          }
        }
      ]
    }
  }
}
```

在前面的示例中，`linear`是`decay`函数的名称。使用它时，值将线性衰退。其他可能的值是`gauss`和`exp`。我们选择了`linear`衰减函数，因为当字段值两次超过给定值时，它把分数设置为`0`。当你想把较低的值赋予太远的文档时，这是非常有用的。

在此给出相关方程，让你了解得分是如何通过给定函数计算的。`linear`衰减函数使用以下公式来计算得分文档：

5

$$score = \max\left(0, \frac{scale - |\,field\ value - origin\,|}{scale}\right)$$

`gauss`衰减函数使用以下公式来计算得分文档：

$$score = \exp\left(-\frac{(field\ value - origin)^2}{2scale^2}\right)$$

`exp`衰减函数使用以下公式来计算得分文档：

$$score = \exp\left(-\frac{|\,field\ value - origin\,|}{scale}\right)$$

当然，你不需要每次都使用纸笔计算文档，但偶尔需要时，这些函数很方便。

现在，让我们讨论一下查询结构的其余部分。我们想用`point`字段作得分计算。如果文档没

有该字段的值，在计算时会得到一个值为1。

此外，我们提供了额外的参数。origin和scale参数是必需的。origin是中心点，计算从此点开始。scale是衰变率。默认情况下，offset参数设置为0；如果定义了该参数，decay函数将只计算文档值比此参数的值大的文档得分。decay参数告诉Elasticsearch应该降低多少分数，默认设置为0.5。在我们的例子中，我们要求，在1公里的距离，得分应该会减少20%（0.2）。

> 我们期望新版的Elasticsearch会扩展可用函数的数量，建议跟进官方文档，function_score查询的专属页面可访问这里：http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl- function-score-query.html。

4. 弃用查询

在介绍function_score查询之后，custom_boost、custom_score和custom_filters_score查询被弃用。以下部分演示如何使用function_score查询实现与这些查询相同的结果。本节为想从Elasticsearch旧版本迁移并通过修改来移除废弃查询的人提供一个参考。

(1) 更换custom_boost_factor查询

假设我们有如下的custom_boost_factor查询：

```
{
  "query" : {
    "custom_boost_factor" : {
      "query": {
        "term" : { "author" : "heller" }
      },
      "boost_factor": 5.0
    }
  }
}
```

为使用function_score查询来取代上述查询，可以使用下列查询：

```
{
  "query" : {
    "function_score" : {
      "query": {
        "term" : { "author" : "heller" }
      },
      "functions" : [
        { "boost_factor": 5.0 }
      ]
    }
  }
}
```

(2) 更换custom_score查询

第二种废弃查询是custom_score，假设有如下的custom_score查询：

```
{
  "query" : {
    "custom_score" : {
      "query" : { "match_all" : {} },
      "script" : "_source.copies * 0.5"
    }
  }
}
```

如果想用function_score查询取代它，将如下所示：

```
{
  "query" : {
    "function_score" : {
      "boost_mode" : "replace",
      "query" : { "match_all" : {} },
      "functions" : [
        {
          "script_score" : {
            "script" : "_source.copies * 0.5"
          }
        }
      ]
    }
  }
}
```

(3) 更换custom_filters_score查询

最后要讨论的是custom_filters_score查询，假设有如下查询：

```
{
  "query" : {
    "custom_filters_score" : {
      "query" : { "match_all" : {} },
      "filters" : [
        {
          "filter" : { "term" : { "available" : true }},
          "boost" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

如果想用function_score查询取代它，将如下所示：

```
{
  "query" : {
    "function_score" : {
      "query" : { "match_all" : {} },
      "functions" : [
        {
          "filter" : { "term" : { "available" : true }},
          "boost_factor" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

5.5　索引时加权何时有意义

前一节讨论了对查询进行加权。这种类型的加权非常方便和强大，在大多数情况下满足需求。然而，如果当我们在建立索引时就知道哪些文档重要，更方便的方法是使用索引时加权。

我们获得独立于查询的一个加权，成本是重建索引（在加权值变化时，我们需要重建索引）。此外，由于加权过程中已经在索引时计算，性能会稍好一些。Elasticsearch把加权的信息存储为规范化信息的一部分。很重要的是，如果把omit_norms设置为true，就不能使用索引时加权。

5.5.1　在输入数据中定义字段加权

我们看一个典型的文档定义，如下所示：

```
{
  "title" : "The Complete Sherlock Holmes",
  "author" : "Arthur Conan Doyle",
  "year" : 1936
}
```

如果想为这个特定文档的author字段加权，结构应该会略有变化，文档看起来应该如下所示：

```
{
  "title" : "The Complete Sherlock Holmes",
  "author" : {
    "_value" : "Arthur Conan Doyle",
    "_boost" : 10.0,
  },
  "year": 1936
}
```

就是这些。在上述文档被编制到索引后，我们会让Elasticsearch知道author字段的重要性大于其他字段。

> 在旧版本Elasticsearch中，设置文档范围的加权是可能的。然而，从4.0开始，Lucene不支持文档范围的加权，Elasticsearch靠加权所有字段来模拟文档的加权。Elasticsearch 1.0废弃了文档提升，我们决定不写它，因为它将被删除。

5.5.2 在映射中定义加权

值得一提的是可以直接在映射文件中定义字段的加权。下面的示例演示了这一点：

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "title" : { "type" : "string" },
        "author" : { "type" : "string", "boost" : 10.0 }
      }
    }
  }
}
```

因为上述加权，所有查询将对以`author`命名的所有字段加权。这也适用于使用`_all`字段的查询。

5.6 同义词

你应该听说过同义词：有相同或相近意思的词语。有时，当一个词输入搜索框时，我们希望其他一些词也被匹配。回忆一下3.1.1节，有本书叫`Crime and Punishment`。如果我们希望这本书不仅在搜索`crime`或`punishment`时匹配到，也可使用`criminality`和`abuse`搜索到，就要使用同义词。

5.6.1 同义词过滤器

为了使用同义词过滤器，我们需要定义自己的分析器，称为`synonym`，使用空格分词器和一个叫`synonym`的过滤器。该过滤器的类型属性必须设置为`synonym`，它告诉Elasticsearch，该过滤器是一个同义词过滤器。此外，我们希望忽略大小写，对大写和小写的同义词一视同仁（设置`ignore_case`属性为`true`）。自定义一个使用同义词过滤器的分析器，需要以下映射：

```
{
  "index" : {
    "analysis" : {
      "analyzer" : {
        "synonym" : {
          "tokenizer" : "whitespace",
```

```
            "filter" : [
              "synonym"
            ]
          }
        },
        "filter" : {
          "synonym" : {
            "type" : "synonym",
            "ignore_case" : true,
            "synonyms" : [
              "crime => criminality"
            ]
          }
        }
      }
    }
  }
```

1. 映射中的同义词

在前面的定义中，我们在映射中指定了发给Elasticsearch的同义词规则。为此，需要添加synonyms属性，这是一个同义词规则的数组。例如，下面映射的一部分定义了一个同义词规则：

```
"synonyms" : [
  "crime => criminality"
]
```

我们马上将讨论如何定义同义词规则。

2. 存储在文件系统中的同义词

Elasticsearch还允许我们使用基于文件的同义词。为使用文件，需要指定synonyms_path属性，而不是synonyms属性。synonyms_path属性应该设成含有同义词定义的文件名，文件路径应该相对于Elasticsearch的config目录。所以，如果我们将同义词保存在config目录的synonyms.txt文件中，为了使用它，应该把synonyms_path设置成synonyms.txt。

如果要使用存储在文件中的同义词，前面映射文件中的同义词过滤器将如下所示：

```
"filter" : {
  "synonym" : {
    "type" : "synonym",
    "synonyms_path" : "synonyms.txt"
  }
}
```

5.6.2　定义同义词规则

到现在为止，我们讨论了要做什么才能在Elasticsearch中使用同义词扩展。现在，让我们看看允许同义词使用哪些格式。

1. 使用Apache Solr同义词

Apache Lucene世界中最常见的同义词结构可能是Apache Solr用的那个，Apache Solr是基于Lucene的一个搜索引擎，就像Elasticsearch一样。这是Elasticsearch处理同义词的默认方法，以下各节讨论定义一个新同义词的可能性。

(1) 显式同义词

简单的映射允许把一个单词列表映射到其他单词。所以，在我们的例子中，如果要criminality映射到crime、abuse映射到punishment，需要定义以下条目：

```
criminality => crime
abuse => punishment
```

当然，一个单词可以映射到多个单词，多个单词也能被映射到单个单词，如下所示：

```
star wars, wars => starwars
```

前面的示例意味着，star wars和wars将被synonym过滤器换成starwars。

(2) 等效同义词

除了显式映射以外，Elasticsearch允许我们使用等效同义词。例如，下面的定义会使所有单词可交换，你可以使用其中一个单词，来匹配包含其中一个单词的文档：

```
star, wars, star wars, starwars
```

(3) 扩展同义词

使用Apache Solr格式的时候，同义词过滤器允许我们使用一个额外的expand属性。当expand属性设置为true（默认为false），Elasticsearch将所有同义词扩大到所有等价形式。假设我们有以下过滤器配置：

```
"filter" : {
  "synonym" : {
    "type" : "synonym",
    "expand": false,
    "synonyms" : [
      "one, two, three"
    ]
  }
}
```

Elasticsearch将把前面的同义词定义映射如下：

```
one, two, thee => one
```

这意味着，one、two、three将被更改为one。如果expand属性设置为true，相同的同义

词定义将以下列方式解释：

```
one, two, three => one, two, three
```

这基本上意味着左侧的每个单词将扩展为右侧的所有单词。

2. 使用WordNet同义词

如果想使用WordNet结构的同义词（学习更多关于WordNet，请访问http://wordnet.princeton.edu），我们需要为同义词过滤器提供额外的`format`属性，应将其值设置为`wordnet`，以便Elasticsearch理解。

5.6.3　查询时或索引时的同义词扩展

与所有分析器一样，你可能会问我们应该在什么时候使用同义词过滤器：在索引期间，在查询期间或者两者皆可？当然，这取决于你的需求。但是记住，如果使用索引时同义词，在每个同义词更改后，需要重新索引数据。那是因为它们需要重新应用到所有文档。如果只使用查询时同义词，我们可以更新同义词列表，在查询时应用。

5.7　理解解释信息

与数据库相比，使用能执行全文搜索的系统往往不那么显而易见。我们可以同时搜索多个字段，在索引中的数据可以和提供的文档字段值不同（因为分析过程、同义词、缩写等）。甚至更糟的是，默认情况下，搜索引擎按照数据的相关性排序：一个表示文档相对于查询的相似性的数字，这里的关键是相似性。我们已经讨论过，得分需要考虑很多因素：在文档中发现了搜索词，词出现的频率，字段中包含多少词条，等等。这看上去很复杂，而且要想知道为什么一个文档被找到，为什么另一个文档“更好”是不容易的。好在Elasticsearch的一些工具可以回答这些问题，我们现在来看看。

5.7.1　理解字段分析

最常见的问题之一是为什么找不到给定文档。在许多情况下，问题在于映射的定义和分析流程的配置。为了调试分析过程，Elasticsearch提供一个专门的REST API端点，`_analyze`。

让我们先看Elasticsearch的默认分析器返回的信息。运行以下命令：

```
curl -XGET 'localhost:9200/_analyze?pretty' -d 'Crime and Punishment'
```

响应中，我们将得到如下数据：

```
{
  "tokens" : [ {
    "token" : "crime",
    "start_offset" : 0,
    "end_offset" : 5,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "punishment",
    "start_offset" : 10,
    "end_offset" : 20,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}
```

我们可以看到，Elasticsearch把输入短语划分为两个标记。处理时，普通词and被省略（因为它在停用词列表中），而其他单词变成了小写。这显示了在分析过程中到底会发生什么。我们也可以提供分析器的名称，例如，可以将前面的命令修改成下面这样：

```
curl -XGET 'localhost:9200/_analyze?analyzer=standard&pretty' -d
  'Crime and Punishment'
```

上述命令使我们能够检查标准分析器是如何分析数据的（会跟我们前面看到的响应有点不同）。

值得注意的是，有另一种分析API的可用形式：我们能够提供分词器和过滤器。要在创建目标映射之前实验一下配置时，它非常方便。调用的示例如下所示：

```
curl -XGET
  'localhost:9200/library/_analyze?tokenizer=whitespace&
    filters=lowercase,kstem&pretty' -d 'John Smith'
```

上面的例子使用了一个分析器，它由whitespace分词器和两个过滤器（lowercase和kstem）组成。

可以看到，分析API可以非常有效地跟踪映射配置中的错误，对解决查询和搜索相关性问题也非常有用。它可以告诉我们分析器如何工作，它们生成的词条是什么，这些词条的属性是什么。有了这些资料，分析查询问题会更容易追踪。

5.7.2 解释查询

除了看在分析期间发生了什么，Elasticsearch让我们可以解释特定的查询和文档是如何计算得分的。看下面的示例：

```
curl -XGET 'localhost:9200/library/book/1/_explain?pretty&q=quiet'
```

上面的命令将返回如下的响应结果：

```
{
  "_index" : "library",
  "_type" : "book",
  "_id" : "1",
  "matched" : true,
  "explanation" : {
    "value" : 0.057534903,
    "description" : "weight(_all:quiet in 0) [PerFieldSimilarity],
      result of:",
    "details" : [ {
      "value" : 0.057534903,
      "description" : "fieldWeight in 0, product of:",
      "details" : [ {
        "value" : 1.0,
        "description" : "tf(freq=1.0), with freq of:",
        "details" : [ {
          "value" : 1.0,
          "description" : "termFreq=1.0"
        } ]
      }, {
        "value" : 0.30685282,
        "description" : "idf(docFreq=1, maxDocs=1)"
      }, {
        "value" : 0.1875,
        "description" : "fieldNorm(doc=0)"
      } ]
    } ]
  }
}
```

看起来很复杂，好吧，确实很复杂！更糟糕的是，这只是一个简单的查询！Elasticsearch，更确切地说，是Lucene库，会显示关于评分过程的内部信息。我们只蜻蜓点水地解释一下上述响应中最重要的东西。

最重要的部分是为文档计算的总分（explanation对象的value属性）。如果等于0，说明文档与给定查询不匹配。另一个重要的元素是description部分，告诉我们采用了哪种相似度模型。在我们的示例中，查询quiet词条，在_all字段被找到。这很直观，因为我们在默认字段中搜索，就是_all（参见2.4节）。

details部分提供了有关组件的信息，以及我们应在哪里寻求解释：为什么我们的文档与查询匹配。说到评分，我们有一个对象：单一的负责文档得分计算的组件。value属性是由此组件计算的得分，然后再次看到description和detail节点。正如你在description字段中看到的，最后的得分（fieldWeight in 0, product of）是内部details数组中的每个元素所计算得分的组成值（1.0 * 0.30685282 * 0.1875）。

在内部的details数组里，我们可以看到3个对象。第一个显示了给定字段上的词频信息（在

我们的例子中是1），这意味着字段上只出现一次搜索词条。第二个对象显示了逆文档频率。注意`maxDocs`属性，它等于`1`，意味着给定词条只找到了1个文档。第三个对象负责字段的`field norm`。

请注意，每个查询的上述响应是不同的。而且，查询越复杂，返回信息也会越复杂。

5.8 小结

本章介绍了Apache Lucene评分在内部如何工作，如何使用Elasticsearch的脚本功能，以及如何索引和搜索不同语言的文档。我们使用了不同的查询来改变文档的得分，并且修改查询来让它适合我们的用例。学到了索引时的加权，什么是同义词，以及它们如何帮助我们。最后，学习了如何检查为什么特定的文档是结果集的一部分，以及得分是如何计算的。

下一章将介绍全文搜索之外的内容。看看聚合是什么，以及如何使用它们来分析数据。我们也会看到切面，它能够聚合数据并为其带来意义。使用建议器执行拼写检查和自动完成，采用前瞻性搜索找到匹配特定查询的文档。索引二进制文档，并使用地理空间功能，使用地理数据搜索。最后，我们将使用卷动API有效获取大量结果，还会看到如何让Elasticsearch在查询中使用词条列表（一个自动加载的列表）。

第 6 章 超越全文检索

上一章介绍了Apache Lucene评分机制的内部工作原理；展示了Elasticsearch的脚本功能，在不同的语言中索引和搜索文档，使用不同的查询来改变文档的得分；使用了索引时加权；学习了什么是同义词，最后，我们看到了检查为什么特定的文档是结果集的一部分，它的得分如何计算。本章主要内容如下：

- 使用聚合来聚合索引数据，并从中计算有用的信息；
- 采用切面来计算不同的统计数据；
- 使用Elasticsearch建议器实现拼写检查和自动完成功能；
- 使用预搜索来匹配文档；
- 索引二进制文件；
- 索引和搜索地理数据；
- 高效获取大数据集；
- 自动加载并在查询中使用词条。

6.1 聚合

Elasticsearch 1.0带来了改进和新功能，更包括备受期待的框架，它赋予Elasticsearch一个新定位：全功能分析引擎。现在，你可以使用Elasticsearch作为各种系统的一个关键部分，处理大量的数据，提取结论，并将这些数据可视化为可读的方式。我们看看如何使用这些功能，以及可以用它们做些什么。

6.1.1 一般查询结构

为了使用聚合（aggregation），需要在查询中增加一个额外节点。一般来说，使用聚合的查询看上去像下面的代码片段：

```
{
  "query": { … },
  "aggs" : { … }
}
```

aggs属性（你可以使用aggregations属性，aggs只是个缩写），可以定义任意数量的聚合。然而要记住一件事，键值定义了聚合的名称（需要它来区分服务器响应中的特定聚合）。我们使用library索引，创建第一个使用聚合的查询。发送如下查询命令：

```
curl 'localhost:9200/_search? Search_type=count & pretty' -d '{
  "aggs": {
    "years": {
      "stats": {
        "field": "year"
      }
    },
    "words": {
      "terms": {
        "field": "copies"
      }
    }
  }
}'
```

此查询定义了两个聚合。名为years的聚合显示year字段的统计信息，words聚合包含给定字段中所使用词条的信息。

> 在示例中，我们假定在搜索的同时聚合。如果不需要搜索到的文档，用search_type=count参数会更好，这将省略一些不必要的工作，因此更高效。在那种情况下，端点应为/library/_search?search_type=count。3.2节介绍了更多关于搜索类型的内容。

现在来看看Elasticsearch为上述查询返回的响应：

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 4,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "words": {
      "buckets": [
        {
          "key": 0,
          "doc_count": 2
        },
```

```
        {
          "key": 1,
          "doc_count": 1
        },
        {
          "key": 6,
          "doc_count": 1
        }
      ]
    },
    "years": {
      "count": 4,
      "min": 1886,
      "max": 1961,
      "avg": 1928,
      "sum": 7712
    }
  }
}
```

你可以看到，两个聚合都被返回。查询中定义的第一个聚合（years）返回了给定字段的一般统计，它是从所有匹配查询的文档中搜集而来的。第二个聚合（words）有点不同，它创建了名为buckets的集合，基于返回文档计算而来，每个聚合值都出现在集合里。可以看到，有多种聚合类型，每种都返回不同的结果。本节后面会展示这些区别。

6.1.2　可用的聚合

看过前面的例子之后，你不应该对聚合分成几组感到惊讶。目前，有两组聚合：度量聚合（metric aggregation）和桶聚合（bucketing aggregation）。

1. 度量聚合

度量聚合接收一个输入文档集并生成至少一个统计值。你将看到，这些聚合大多是自解释的。

(1) min、max、sum、avg聚合

min、max、sum和avg聚合的使用很相似。它们对于给定字段分别返回最小值、最大值、总和和平均值。任何数值型字段都可以作为这些值的源。比如，为了计算year字段的最小值，可以构建如下聚合：

```
{
  "aggs": {
    "min_year": {
      "min": {
        "field": "year"
      }
    }
  }
}
```

返回值类似于如下：

```
"min_year": {
  "value": 1886
}
```

(2) 使用脚本

输入值也可以由脚本生成。如果想找到year字段所有值的最小值，且要从这些值中减去1000，我们将发送如下聚合：

```
{
  "aggs": {
    "min_year": {
      "min": {
        "script": "doc['year'].value - 1000"

      }
    }
  }
}
```

在本例中，聚合使用的值是原始year字段值减去1000。可以用另外一个写法来得到相同的响应：提供script属性和字段名，如下所示：

```
{
  "aggs": {
    "min_year": {
      "min": {
        "field": "year",
        "script": "_value - 1000"
      }
    }
  }
}
```

在script属性以外提供了字段名，可以写得更详细，如下所示：

```
{
  "aggs": {
    "min_year": {
      "min": {
        "field": "year",
        "script": "_value - mod",
        "params": {
          "mod" : 1000
        }
       }
    }
  }
}
```

可以看到，我们添加了params节点作为附加参数。可以在5.2节中阅读更多关于脚本的内容。

6

(3) value_count聚合

value_count聚合跟前面描述的聚合类似，只是输入字段不一定要是数值型的。该聚合的一个例子如下所示：

```
{
  "aggs": {
    "number_of_items": {
      "value_count": {
        "field": "characters"
      }
    }
  }
}
```

我们暂停一会。这是个很好的机会，来看看这个例子中Elasticsearch聚合计算了哪个值。在library索引的books类型上执行上述查询，响应将如下所示：

```
"number_of_items": {
  "value": 31
}
```

Elasticsearch从所有文档的characters字段计算所有标记。这个数字是有意义的，比如，Sofia Semyonovna Marmeladova这个词条在分析之后将变成sofia、semyonovna和marmeladova。在很多情况下，我们不想要这个行为。为此，应该使用characters字段的未经分析版本。

(4) stats和extended_stats聚合

stats和extended_stats聚合可以看成是在单一聚合对象中返回所有前面描述聚合的一种聚合。如果想计算year字段的统计，可以使用如下代码：

```
{
  "aggs": {
    "stats_year": {
      "stats": {
        "field": "year"
      }
    }
  }
}
```

Elasticsearch返回的相关结果将如下所示：

```
"stats_year": {
   "count": 4,
   "min": 1886,
   "max": 1961,
   "avg": 1928,
   "sum": 7712
}
```

当然，extended_stats聚合返回的统计数据包含更多扩展信息。看看下面的查询：

```
{
  "aggs": {
    "stats_year": {
      "extended_stats": {
        "field": "year"
      }
    }
  }
}
```

在返回的响应中，可以看到如下输出：

```
"stats_year": {
   "count": 4,
   "min": 1886,
   "max": 1961,
   "avg": 1928,
   "sum": 7712,
   "sum_of_squares": 14871654,
   "variance": 729.5,
   "std_deviation": 27.00925767213901
}
```

可以看到，除了已知的值，我们还得到平方和、方差和标准差统计。

2. 桶聚合

桶聚合返回很多子集，并限定输入数据到一个特殊的叫做桶的子集中。可以把桶聚合想象成类似前面切面功能的东西，6.2节有介绍。然而，这个聚合更强大、更易于使用。我们来简单介绍一遍可用的桶聚合。

(1) terms聚合

terms聚合为字段中每个词条返回一个桶。这允许你生成字段每个值的统计。例如，通过这个聚合，可以回答以下问题。

- 每年出版多少书？
- 多少书可以用来出借？
- 我们最多的书有多少册？

对上面最后一个问题，可以发送如下查询：

```
{
  "aggs": {
    "availability": {
      "terms": {
        "field": "copies"
```

```
        }
      }
    }
  }
```

Elasticsearch对`library`索引的响应如下所示：

```
"availability": {
  "buckets": [
    {
      "key": 0,
      "doc_count": 2
    },
    {
      "key": 1,
      "doc_count": 1
    },
    {
      "key": 6,
      "doc_count": 1
    }
  ]
}
```

我们看到两本书只有一本（`key`属性等于`0`的桶），一本书有另外一本（`key`属性等于`1`的桶），一本书有另外6本（`key`属性等于`6`的桶）。默认情况下，Elasticsearch返回的桶按照`doc_count`的值倒序排序。我们可以通过添加`order`属性来改变。比如，为了使用key属性值对聚合排序，可以发送以下查询：

```
{
  "aggs": {
    "availability": {
      "terms": {
        "field": "copies",
        "size": 40,
        "order": { "_term": "asc" }
      }
    }
  }
}
```

可以按升序排（`asc`），也可以按降序排（`desc`）。在我们的例子中，使用key属性（`_team`）排序。另一个选择是`_count`，告诉Elasticsearch使用`doc_count`属性来排序。

前面的例子中，我们还添加了`size`属性。你可以猜到它定义了最多返回多少个桶。

> 你应该记得，当字段被分析时，会从如例子所示的分析词条中得到桶和计数值。这可能不是你想要的答案，解决办法就是在索引中为字段添加一个额外的、未经分析的版本，并使用它来做聚合计算。

(2) `range`聚合

`range`聚合使用定义的范围来创建桶。如果我们想检查给定时间区间有多少书出版，可以创建如下查询：

```
{
  "aggs": {
    "years": {
      "range": {
        "field": "year",
        "ranges": [
          { "to" : 1850 },
          { "from": 1851, "to": 1900 },
          { "from": 1901, "to": 1950 },
          { "from": 1951, "to": 2000 },
          { "from": 2001 }
        ]
      }
    }
  }
}
```

对`library`索引中的数据，响应将如下所示：

```
"years": {
  "buckets": [
    {
      "to": 1850,
      "doc_count": 0
    },
    {
      "from": 1851,
      "to": 1900,
      "doc_count": 1
    },
    {
      "from": 1901,
      "to": 1950,
      "doc_count": 2
    },
    {
      "from": 1951,
      "to": 2000,
      "doc_count": 1
    },
    {
      "from": 2001,
      "doc_count": 0
    }
  ]
}
```

从上面的输出可以看出，1901~1950年，我们出版了2本书。

创建用户界面时，可以为每个桶自动生成一个标签。打开此功能很简单：只需要添加keyed属性并将其设置为true，就像下面的示例：

```
{
  "aggs": {
    "years": {
      "range": {
        "field": "year",
        "keyed": true,
        "ranges": [
          { "to" : 1850 },
          { "from": 1851, "to": 1900 },
          { "from": 1901, "to": 1950 },
          { "from": 1951, "to": 2000 },
          { "from": 2001 }
        ]
      }
    }
  }
}
```

上述代码中突出显示的部分，使结果中包含标签，在Elasticsearch的以下响应中可以看到：

```
"years": {
  "buckets": {
    "*-1850.0": {
      "to": 1850,
      "doc_count": 0
    },
    "1851.0-1900.0": {
      "from": 1851,
      "to": 1900,
      "doc_count": 1
    },
    "1901.0-1950.0": {
      "from": 1901,
      "to": 1950,
      "doc_count": 2
    },
    "1951.0-2000.0": {
      "from": 1951,
      "to": 2000,
      "doc_count": 1
    },
    "2001.0-*": {
      "from": 2001,
      "doc_count": 0
    }
  }
}
```

你可能已经注意到，结构略有变化，buckets字段不再是表，而是图，键值是从范围生成的。这行得通，但不太漂亮。我们的例子中，给每个桶一个名称会更有用。这是可能的，我们可以为每个范围添加key属性并把它的值设置为所需的名称。思考以下示例：

```
{
  "aggs": {
    "years": {
      "range": {
        "field": "year",
        "keyed": true,
        "ranges": [
          { "key": "Before 18th century", "to": 1799 },
          { "key": "18th century", "from": 1800, "to": 1899 },
          { "key": "19th century", "from": 1900, "to": 1999 },
          { "key": "After 19th century", "from": 2000 }
        ]
      }
    }
  }
}
```

> 有一点很重要也很有用，范围可以不叠加。在这种情况下，Elasticsearch将正确计数多桶文档。

(3) date_range聚合

date_range聚合类似于前面讨论的range聚集，但它专用在使用日期类型的字段。虽然library索引文档中有year字段，但这个字段是一个数字，不是日期。为了测试，假设我们希望扩展图书馆索引来支持报纸，使用以下命令创建一个新的library2索引：

```
curl -XPOST localhost:9200/_bulk --data-binary '{ "index": {"_index":
  "library2", "_type": "book", "_id": "1"}}
{ "title": "Fishing news", "published": "2010/12/03 10:00:00",
  "copies": 3, "available": true }
{ "index": {"_index": "library2", "_type": "book", "_id": "2"}}
{ "title": "Knitting magazine", "published": "2010/11/07 11:32:00",
  "copies": 1, "available": true }
{ "index": {"_index": "library2", "_type": "book", "_id": "3"}}
{ "title": "The guardian", "published": "2009/07/13 04:33:00",
  "copies": 0, "available": false }
{ "index": {"_index": "library2", "_type": "book", "_id": "4"}}
{ "title": "Hadoop World", "published": "2012/01/01 04:00:00",
  "copies": 6, "available": true }
'
```


在这个索引中，没有使用映射作为Elasticsearch发现机制，这对我们的例子来说足够了。我们开始第一个使用date_rage聚合的查询，如下所示：

```
{
```

```
  "aggs": {
    "years": {
      "date_range": {
        "field": "published",
        "ranges": [
          { "to" : "2009/12/31" },
          { "from": "2010/01/01", "to": "2010/12/31" },
          { "from": "2011/01/01" }
          ]
        }
    }
  }
}
```

与普通的range聚合比较，唯一改变的是聚合类型（date_range）。可以用Elasticsearch认可的字符串格式传递日期（更多信息请参阅第2章），或者用数值：自1970-01-01以来的毫秒数。Elasticsearch返回的响应如下：

```
"years": {
  "buckets": [
    {
      "to": 1262217600000,
      "to_as_string": "2009/12/31 00:00:00",
      "doc_count": 1
    },
    {
      "from": 1262304000000,
      "from_as_string": "2010/01/01 00:00:00",
      "to": 1293753600000,
      "to_as_string": "2010/12/31 00:00:00",
      "doc_count": 2
    },
    {
      "from": 1293840000000,
      "from_as_string": "2011/01/01 00:00:00",
      "doc_count": 1
    }
  ]
}
```

上述响应与之前range聚合的响应相比，唯一的区别是，关于范围边界的信息分为了两个属性。from或to属性表示自1970-01-01以来的毫秒数。from_as_string和to_as_string属性以人类可读的形式表示日期。当然，date_range聚合中定义的keyed和key属性照常工作。

Elasticsearch也允许使用format属性来定义日期格式。在我们的示例中，以年的形式表达日期，所以天数和时间是没有必要的。如果想显示月份名称，可以发送如下查询：

```
{
  "aggs": {
    "years": {
      "date_range": {
```

```
          "field": "published",
          "format": "MMMM YYYY",
          "ranges": [
            { "to" : "2009/12/31" },
            { "from": "2010/01/01", "to": "2010/12/31" },
            { "from": "2011/01/01" }
          ]
        }
      }
    }
  }
```

返回的范围之一看上去如下所示：

```
{
  "from": 1262304000000,
  "from_as_string": "January 2010",
  "to": 1293753600000,
  "to_as_string": "December 2010",
  "doc_count": 2
}
```

看上去好多了，对吧？

> format参数中可以使用的格式定义在Joda Time库中，完整列表请参阅http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html。

关于date_range聚合，还有一点。有时，我们会想要建立一个能随时间变化的聚合，例如想看到每个季度出版了多少报纸。不修改查询也可以做到，请思考以下示例：

```
{
  "aggs": {
    "years": {
      "date_range": {
        "field": "published",
        "format": "dd-MM-YYYY",
        "ranges": [
          { "to" : "now-9M/M" },
          { "to" : "now-9M" },
          { "from": "now-6M/M", "to": "now-9M/M" },
          { "from": "now-3M/M" }
        ]
      }
    }
  }
}
```

这里的关键是如now-9M的表达式。Elasticsearch使用数学生成相应的值。你可以使用y（年）、M（月）、w（周）、d（日）、h（小时）、m（分钟）和s（秒）。例如，表达式now+3d表示现在起的3天后。在我们的示例中，/M表示只取已被转成月份的日期。由于这种表示法，我们可以只计

算完整月。第二个优点是计算的日期对缓存更友好，如果没有四舍五入，日期每一毫秒都更改，导致居于range的每个缓存都没有意义。

(4) IPv4 range聚合

range聚合的最后一个形式是基于互联网地址的聚合。它工作在定义成ip类型的字段上，允许以CIDR的格式来定义IP范围（CIDR：http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing）。ip_range聚合的示例如下所示：

```
{
  "aggs": {
    "access": {
      "ip_range": {
        "field": "ip",
        "ranges": [
          { "from": "192.168.0.1", "to": "192.168.0.254" },
          { "mask": "192.168.1.0/24" }
        ]
      }
    }
  }
}
```

上述查询的响应如下所示：

```
"access": {
  "buckets": [
    {
      "from": 3232235521,
      "from_as_string": "192.168.0.1",
      "to": 3232235774,
      "to_as_string": "192.168.0.254",
      "doc_count": 0
    },
    {
      "key": "192.168.1.0/24",
      "from": 3232235776,
      "from_as_string": "192.168.1.0",
      "to": 3232236032,
      "to_as_string": "192.168.2.0",
      "doc_count": 4
    }
  ]
}
```

同样，keyed和key属性跟range聚合中一样。

(5) missing聚合

回到library索引，看看多少条目没有定义原始标题（otitle字段）。为此，我们使用missing聚合，在这种情况下它是个好东西。示例查询如下所示：

```
{
  "aggs": {
    "missing_original_title": {
      "missing": {
        "field": "otitle"
      }
    }
  }
}
```

响应中相关部分如下所示：

```
"missing_original_title": {
  "doc_count": 2
}
```

说明我们有2个文档没有otitle字段。

> missing聚合提示映射定义可能定义了null_value，需要独立于这个定义来对文档计数。

(6) nested聚合

4.3节介绍了嵌套文档。使用这个数据来看看下一种聚合类型：nested聚合。我们来创建一个最简单的可工作查询，如下所示：

```
{
  "aggs": {
    "variations": {
      "nested": {
        "path": "variation"
      }
    }
  }
}
```


上面的查询在结构上类似于任何其他聚合。它包含参数path，指向嵌套的文件。在响应中，我们得到一个数字，如以下输出所示：

```
"variations": {
  "doc_count": 2
}
```

上述响应意味着索引中有两个嵌套文档使用提供的variation类型。

(7) histogram聚合

histogram聚合定义桶。使用该聚合的最简单查询如下所示：

```
{
  "aggs": {
```

```
    "years": {
      "histogram": {
        "field" : "year",
        "interval": 100
      }
    }
  }
}
```

这里，新的信息片段是interval，它定义了将用于创建桶的每个范围的长度。因此，在我们的示例中，桶将以100年为周期来创建。把上面查询发送到我们的library索引，响应中聚合部分如下所示：

```
"years": {
  "buckets": [
    {
      "key": 1800,
      "doc_count": 1
    },
    {
      "key": 1900,
      "doc_count": 3
    }
  ]
}
```

与range聚合一样，histogram聚合同样允许我们使用keyed属性。其他可用的选项是min_doc_count，使我们能够控制为创建一个桶需要的最小文档数目。如果把min_doc_count属性设置为零，Elasticsearch还将包括文档数目为0的桶。

(8) date_histogram聚合

正如date_range聚合是range聚合的一种特殊形式，date_histogram聚合也是histogram聚合的一种扩展，专用在日期上。所以我们再次使用有报纸的索引（名字为library2）。使用date_histogram聚合的查询示例如下所示：

```
{
  "aggs": {
    "years": {
      "date_histogram": {
        "field" : "published",
        "format" : "yyyy-MM-dd HH:mm",
        "interval": "10d"
      }
    }
  }
}
```

可以看到interval属性上一个重要的区别。它现在用一个字符串来描述时间间隔，在我们的例子中是10天。当然，可以将它设置为任何值，使用与在date_range聚合中讨论过的格式相

同的后缀。值得一提的是，数字可以是一个浮点值，例如1.5m，即每1.5分钟。format属性跟date_range聚合中一样，归功于此，Elasticsearch可以根据定义的格式添加一个人类可读的日期。当然，format属性不是必需的，但它是有用的。除此以外，类似于range聚合，keyed和min_doc_count属性仍然有效。

(9) 时区

Elasticsearch将所有日期存储成UTC时区。你可以定义用于显示的时区。日期转换有两种方法，可以在把元素分配给桶之前转换日期，也可以在分配之后转换。因此，取决于所选方法和桶的定义，一个元素可能分配给不同的桶。有两个属性定义此行为：pre_zone和post_zone。此外，还有一个time_zone，基本上用来设置pre_zone属性的值。有如下三种符号来设置这些属性。

- 可以设置小时偏移，例如：pre_zone:-4或time_zone:5；
- 可以使用时间格式，例如：pre_zone:"-4:30"；
- 可以使用时区的名字，例如：time_zone:"Europe/Warsaw"。

> http://joda-time.sourceforge.net/timezones.html可以看到所有可用的时区。

(10) geo_distance聚合

下来的两个聚合与地图和空间搜索有关。本章后面部分会谈到地理类型和查询，所以现在可以跳过这两个主题，稍后再回来看。

看看下面的查询：

```
{
  "aggs": {
    "neighborhood": {
      "geo_distance": {
        "field": "location",
        "origin": [-0.1275, 51.507222],
        "ranges": [
          { "to": 1200 },
          { "from": 1201 }
        ]
      }
    }
  }
}
```

你可以看到，这类似于range聚合查询，计算有多少个城市分别落在两个桶内：第一个桶是在1200公里内的城市，第二个桶是1200公里以外的城市（在我们的例子中，起源是伦敦）。Elasticsearch返回的响应中，聚合部分看起来如下：

```
"neighborhood": {
  "buckets": [
    {
      "key": "*-1200.0",
      "from": 0,
      "to": 1200,
      "doc_count": 1
    },
    {
      "key": "1201.0-*",
      "from": 1201,
      "doc_count": 4
    }
  ]
}
```

当然，在geo_distance聚合中，keyed和key属性仍然有效。

现在，修改上述查询，展现geo_distance聚合的其他可能性，如下所示：

```
{
  "aggs": {
    "neighborhood": {
      "geo_distance": {
        "field": "location",
        "origin":    { "lon": -0.1275, "lat": 51.507222},
        "unit": "m",
        "distance_type" : "plane",
          "ranges": [
            { "to": 1200 },
            { "from": 1201 }
          ]
      }
    }
  }
}
```

在上面的查询我们高亮了三件事。第一个变化是关于如何定义原点。可以通过多种形式指定位置，这将在本章后面关于地理类型中更准确地描述。

第二个变化是unit属性。可能的值有km（默认）、mi、in、yd、m、cm和mm，它们定义了数字的单位（分别是公里、英里、英寸、码、米、厘米和毫米）。

最后一个属性distance_type，指定了Elasticsearch如何计算距离。从最快但精度最低到最慢但精度最高，可能的值有：plane、sloppy_arc（默认值）和arc。

(11) geohash_grid聚合

现在你知道了如何用与给定点的距离做聚合。第二个选择是把区域组织成一个网格，并分配每个位置到相应的单元格中。为此，理想的方案是地理散列（Geohash，http://en.wikipedia.

org/wiki/Geohash），它把位置编码成一个字符串。

字符串越长，对特定位置的描述越精确。比如，一个字母足以描述一个5000 × 5000平方公里的盒子，而五个字母足够描述约5 × 5平方公里的广场。让我们看看下面的查询：

```
{
  "aggs": {
    "neighborhood": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      }
    }
  }
}
```

我们用提到的5 × 5平方公里广场的精度来定义geohash_grid聚合，precision属性描述了用在geohash字符串对象中的字母长度。关于分辨率与geohash长度的表，可以在这个网址找到：http://www.elasticsearch.org/guide/en/elasticsearch/reference/master/search-aggregations-bucket-geohashgrid-aggregation.html。

当然，更精确通常意味着对系统更大的压力，因为需要更多的桶。默认情况下，Elasticsearch不会生成一万个以上的桶。可以使用size属性提高此参数，但事实上，你应该尽可能减少它。

6.1.3 聚合的嵌套

这是一个强大的功能，使我们能够构建复杂的查询。用nested聚合扩展一个例子。在使用nested聚合的例子中，我们只可能在嵌套文档中工作。但是，看看下面的示例，就知道添加nested聚合时会发生什么：

```
{
  "aggs": {
    "variations": {
      "nested": {
        "path": "variation"
      },
      "aggs": {
        "sizes": {
          "terms": {
            "field": "variation.size"
          }
        }
      }
    }
  }
}
```

可以看到，我们添加了一个嵌套的聚合，叫做`sizes`。上述查询结果的聚合部分如下所示：

```
"variations": {
  "doc_count": 2,
  "sizes": {
    "buckets": [
      {
        "key": "XL",
        "doc_count": 1
      },
      {
        "key": "XXL",
        "doc_count": 1
      }
    ]
  }
}
```

完美！Elasticsearch从父聚合中取得结果，并对其使用`terms`聚合来分析。聚合甚至可以进一步嵌套，理论上，可以无限嵌套聚合。也可以在同一级别上有更多的聚合。

看看下面的例子：

```
{
  "aggs": {
    "years": {
      "range": {
        "field": "year",
        "ranges": [
          { "to" : 1850 },
          { "from": 1851, "to": 1900 },
          { "from": 1901, "to": 1950 },
          { "from": 1951, "to": 2000 },
          { "from": 2001 }
        ]
      },
      "aggs": {
        "statistics": {
          "stats": {}
        }
      }
    }
  }
}
```

你可能会看到前面的示例类似于讨论`range`聚合时用到的一个例子。然而，现在我们添加了一个额外的聚合，给每个桶增加了统计。其中一个桶的输出将如下所示：

```
{
  "from": 1851,
  "to": 1900,
  "doc_count": 1,
```

```
    "statistics": {
      "count": 1,
      "min": 1886,
      "max": 1886,
      "avg": 1886,
      "sum": 1886
    }
  }
```

注意，在stats聚合中，我们省略了用于计算统计的有关字段的信息。Elasticsearch足够聪明，可以从上下文中获得此信息：在这个例子中，是父聚合。

6.1.4 桶排序和嵌套聚合

让我们回忆terms聚合和排序的例子。我们说到，可以在桶的键值或者文档的数量上排序。这只是部分正确，Elasticsearch还可以使用嵌套聚合里的值！让我们用以下查询示例作为开始：

```
{
  "aggs": {
    "availability": {
      "terms": {
        "field": "copies",
         "order": { "numbers.avg": "desc" }
        },
        "aggs": {
          "numbers": { "stats" : {}}
      }
    }
  }
}
```

在前面的示例中，availability聚合的顺序基于numbers聚合的平均值。在这个例子中，numbers.avg符号是必需的，因为stats聚合是多值的。如果是sum聚合，只要聚合的名称就足够了。

6.1.5 全局和子集

我们所有的例子都有一个共同点：聚合考虑了整个索引中的数据。聚合框架允许我们操作由查询返回的过滤文档，或相反，完全忽略查询。你还可以混合使用这两种方法。来分析以下示例：

```
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
    },
    "filter": {
      "term": {
```

```
            "available": "true"
          }
        }
      }
    },
    "aggs": {
      "with_global": {
        "global": {},
        "aggs": {
          "copies": {
            "value_count": {
              "field": "copies"
            }
          }
        }
      },
      "without_global": {
        "value_count": {
          "field": "copies"
        }
      }
    }
  }
```

第一个节点是查询。在这个例子中，我们要返回目前所有可用的书。在第二个节点，看到被命名为with_global和without_global的聚合。这些集合是相似的，都使用value_count对copies字段聚合。不同的是，with_global聚合嵌套在global聚合中。这是一种新的东西：global聚合创建一个包含当前搜索范围中的所有文档的桶（这意味着我们用于搜索的所有索引和类型），但忽略定义的查询。换句话说，global聚合了所有文档，而without_global会使聚合只在由查询返回的文档上工作。

上述查询的响应中，聚合节点看上去如下所示：

```
"aggregations": {
  "without_global": {
    "value": 2
  },
  "with_global": {
    "doc_count": 4,
    "copies": {
      "value": 4
    }
  }
}
```

在我们的索引中，我们有两个文档与查询匹配（目前可用的书）。without_global对这些文档做聚合，得出一个值等于2。with_global聚合忽略搜索操作，操作于索引中的每个文档，意味着所有的四本书。

现在，让我们看看如何建立几个聚合，并且让其中一个对文档的子集进行操作。为此，可以

在聚合中使用过滤器，这将创建一个桶，其中包含的文档被给定的过滤器减少。看看下面的示例：

```
{
  "aggs": {
    "with_filter": {
      "filter": {
        "term": {
          "available": "true"
        }
      },
      "aggs": {
        "copies": {
          "value_count": {
            "field": "copies"
          }
        }
      }
    },
    "without_filter": {
      "value_count": {
        "field": "copies"
      }
    }
  }
}
```

我们没有使用查询来缩小传递到聚合的文档数量，但包括了一个过滤器用来减少将被计算聚合的文档，效果跟我们以前所示一样。

包含和排除

`terms`聚合有另一个可能性来减少聚合数量：可以应用在字符串值上的include/exclude功能，看看下面的查询：

```
{
  "aggs": {
    "availability": {
      "terms": {
        "field": "characters",
        "exclude": "al.*",
        "include": "a.*"
      }
    }
  }
}
```

上述查询运行在一个正则表达式上。它排除了所有以`al`开头的词条，但包含所有以a开头的词条。这样一个查询的影响是，只有以字母a开头的词条才会被计算在内，不包括那些第二个字母是l的单词。正则表达式根据JAVA API定义（http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html），Elasticsearch同样允许你使用定义在这个规格中的`flags`属性。

6

6.2　切面

Elasticsearch是全文搜索引擎，旨在对查询提供搜索结果。然而，有时我们想得到更多，例如根据得到的结果集聚合的数据，价格100~200美元之间的文档数目，或结果文档中最常用的标记。6.1节介绍了聚合框架。除此以外，Elasticsearch提供一个切面模块，负责提供我们提到的那些功能。本章将讨论Elasticsearch提供的不同切面方法。

> 注意，切面提供了聚合模块功能的一个子集。正因为如此，Elasticsearch创作者希望所有用户从切面迁移到上述的聚合模块。切面没有废弃，你还可以使用，但要小心将来它可能从Elasticsearch中移除。

6.2.1　文档结构

为了讨论切面，我们为文档使用一个非常简单的索引结构。它将包含文档的标识符、文档日期、一个用于描述文档的多值字段（`tags`字段），以及一个存有数值信息的字段（`total`字段）。我们的映射看起来如下所示：

```
{
  "mappings" : {
    "doc" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes" },
        "date" : { "type" : "date", "store" : "no" },
        "tags" : { "type" : "string", "store" : "no", "index" :
          "not_analyzed" },
        "total" : { "type" : "long", "store" : "no" }
      }
    }
  }
}
```

> 请记住，在处理字符串字段时，应该避免在分析字段上做切面操作。这种结果可能不是人类可读的，尤其是在使用词干提取，或任何其他重型分析器或过滤器时。

6.2.2　返回的结果

进入如何执行带切面的查询之前，让我们来看看从Elasticsearch切面请求中期待返回的结果。在大多数情况下，你只会对特定切面类型的数据感兴趣。然而，在大多数切面类型中，除了给定

切面类型的特定信息以外，你会还得到以下信息。

- _type：定义使用的切面类型，每个切面类型都将提供此参数。
- missing：定义多少文档没有足够的数据（比如缺乏字段）来计算切面。
- total：定义切面计算中存在的标记数量。
- other：定义未被包含进返回计数的切面值（比如，terms切面中的词条）的数量。

除了这个信息以外，你会得到一个数组，包含关于词条、查询或空间距离的计算切面，如count。例如，下面的代码片段显示通常的切面结果：

```
{
  (...)
  "facets" : {
    "tags" : {
      "_type" : "terms",
      "missing" : 54715,
      "total" : 151266,
      "other" : 143140,
      "terms" : [ {
        "term" : "test",
        "count" : 1119
      }, {
        "term" : "personal",
        "count" : 1063
      },
      (...)
      ]
    }
  }
}
```

可以在结果中看到，切面运行在tags字段上。我们共有151 266个标记被切面模块处理，143 140个标记未包含在结果中。我们也有54 715的文档在tags字段上没有值。test词条出现在1119个文档中，personal词条出现在1063个文档中。这些是你可以预期从切面得到的响应。

6.2.3 使用查询进行切面计算

查询是最简单的切面类型，它允许在切面结果中得到与查询匹配的文档数量。查询本身可以使用Elasticsearch查询语言来表达，我们已经讨论过了。当然，可以包含多个查询来获得多项切面结果。例如，一个为简单的词条查询返回文档数目的切面，代码如下所示：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "my_query_facet" : {
      "query" : {
        "term" : { "tags" : "personal" }
```

```
        }
      }
    }
  }
```

简单的词条查询中包含了一个query类型的切面。前面查询的响应示例如下：

```
{
  (...)
  "facets" : {
    "my_query_facet" : {
      "_type" : "query",
      "count" : 1081
    }
  }
}
```

在响应中你可以看到切面的类型和查询匹配的文档数量，当然，响应中有主要的查询结果，我们省略了。

6.2.4　使用过滤器进行切面计算

除了使用查询以外，Elasticsearch允许使用过滤器来计算切面。这与查询切面非常相似，只是不使用查询，而是使用过滤器。过滤器本身可以使用Elasticsearch查询DSL，并且在单个请求可以使用多个过滤器切面。例如，一个为简单词条过滤器返回文档数目的切面如下所示：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "my_filter_facet" : {
      "filter" : {
        "term" : { "tags" : "personal" }
      }
    }
  }
}
```

你可以看到，我们在简单的词条过滤器中包括了filter类型的切面。谈到性能，filter切面的速度比query切面或包装查询的filter切面更快。

前面查询的响应示例如下：

```
{
  (...)
  "facets" : {
    "my_filter_facet" : {
      "_type" : "filter",
      "count" : 1081
    }
  }
}
```

在响应中我们得到了切面类型，以及与切面过滤器和主查询匹配的文档的数量。

6.2.5　terms切面

切面允许我们指定一个字段，Elasticsearch使用这个字段并返回该字段上最频繁的词条。例如，要计算tags字段上最频繁的词条，可以运行以下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "tags_facet_result" : {
      "terms" : {
        "field" : "tags"
      }
    }
  }
}
```

Elasticsearch将为上述查询返回如下的切面响应：

```
{
  (...)
  "facets" : {
    "tags_facet_result" : {
      "_type" : "terms",
      "missing" : 54716,
      "total" : 151266,
      "other" : 143140,
      "terms" : [ {
        "term" : "test",
        "count" : 1119
      }, {
        "term" : "personal",
        "count" : 1063
      }, {
        "term" : "feel",
        "count" : 982
      }, {
        "term" : "hot",
        "count" : 923
      },
      (...)
      ]
    }
  }
}
```

可以看到，我们在tags_facet_result节点中得到了terms切面结果以及其他已经描述过的信息。

terms切面还可以使用几个额外的参数，如下所示。

- size：此参数指定最多返回多少个最频繁的词条。后面词条的文档数目将包含在结果的other字段中。
- shard_size：此参数指定执行查询的节点上每个分片会获取多少结果。在字段上的独特词条数目大于size参数值的情况下，它允许你提高terms切面的精确度。一般来说，size参数越大，结果越准确，但是计算更昂贵，返回给客户端的数据也更多。为了避免返回一个很长的结果列表，可以把shard_size设置为一个比size参数值大的值。这会告诉Elasticsearch使用它来计算terms切面，但仍然会返回一个不超过size大小的词条。请记住，shard_size参数不能设置成小于size参数的值。
- order：此参数指定该切面的顺序。可设置成：count（默认值，按照频率排序，从最高频率开始）、term（按字母升序顺序）、reverse_count（按照频率排序，从最小频率开始）、reverse_term（按字母降序排序）。
- all_terms：此参数设置为true时，将返回所有词条，即使是那些不匹配任何文档的词条。它可能对性能有要求，特别是对词条很多的字段。
- exclude：此参数指定一组词条，它们将从切面计算中排除。
- regex：此参数指定一个正则表达式，用来控制哪些词条应该包含在计算中。
- script：此参数指定一个脚本，用于在切面计算中处理词条。
- fields：此参数指定一个数组，用于指定切面计算中的多个字段（使用该参数时不再使用field参数）。Elasticsearch将返回跨多个字段的聚合。此属性也可以包括一个特殊值，称为_index。如果该值存在，则按照每个索引返回计算值，因此我们能够区分来自多个索引的切面计算值（当我们对多个索引执行查询时）。
- _script_field：此参数定义了一个脚本，该脚本提供用于计算的实际词条。例如，可以使用基于_source字段的词条。

6.2.6　基于范围的切面

基于范围的切面使我们可以根据定义好的一组范围获取文档数，此外，还可以获取指定字段的汇总数据。如果我们想获得total字段落入小于90、从90到180、大于等于180等范围的文档数量（注意范围的下界是包含的，上界则不含。从90到180意味着>=90且<180），发送以下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "ranges_facet_result" : {
      "range" : {
        "field" : "total",
        "ranges" : [
          { "to" : 90 },
          { "from" : 90, "to" : 180 },
          { "from" : 180 }
        ]
      }
```

```
    }
  }
}
```

在上面的查询中可以看到，我们利用`field`属性来定义字段的名字，`ranges`属性来定义一组范围。每个范围通过使用`from`、`to`或两者来定义。上述查询的响应如下所示：

```
{
  (...)
  "facets" : {
    "ranges_facet_result" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 90.0,
        "count" : 18210,
        "min" : 0.0,
        "max" : 89.0,
        "total_count" : 18210,
        "total" : 39848.0,
        "mean" : 2.1882482152663374
      }, {
        "from" : 90.0,
        "to" : 180.0,
        "count" : 159,
        "min" : 90.0,
        "max" : 178.0,
        "total_count" : 159,
        "total" : 19897.0,
        "mean" : 125.13836477987421
      }, {
        "from" : 180.0,
        "count" : 274,
        "min" : 182.0,
        "max" : 57676.0,
        "total_count" : 274,
        "total" : 585961.0,
        "mean" : 2138.543795620438
      } ]
    }
  }
}
```

可以看到，因为在查询中为范围切面定义了三种范围，我们得到三个范围的响应。每个范围返回以下统计信息。

- `from`：定义了范围的左界（如果在查询中指定）。
- `to`：定义了范围的右界（如果在查询中指定）。
- `min`：定义了给定范围内，用作切面的字段的最小值。
- `max`：定义了给定范围内，用作切面的字段的最大值。
- `count`：定义了落在指定范围内的字段值的文档数目。

6

- total_count：定义了落在指定范围内的字段值的文档总数（对于单值字段来说，应该和count相等，但对于多值字段来说，可能不相等）。
- total：定义了落在指定范围内的所有字段值的总和。
- mean：定义了落在指定范围内用作range切面的字段值的均值。

为聚合数据计算选择不同的字段

如果计算聚合数据统计的字段跟计算范围的字段不一样，我们可以用两个属性：key_field和key_value（或key_script和value_script，允许使用脚本）。key_field属性指定哪个字段用于检查值是否属于给定范围，value_field属性指定哪个字段的值用于聚合计算。

6.2.7　数值和日期直方图切面

直方图切面允许你根据字段值的间隔来建立一个字段值的直方图（对数值型和日期型字段）。如果我们想看多少文档的total字段落到1000的间隔，执行如下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "total_histogram" : {
      "histogram" : {
        "field" : "total",
       "interval" : 1000
      }
    }
  }
}
```

你可以看到，我们使用histogram切面类型，并且除了field属性，还有interval属性，定义了我们想要使用的时间间隔。前面查询的响应示例如下所示：

```
{
  (...)
  "facets" : {
    "total_histogram" : {
      "_type" : "histogram",
      "entries" : [ {
        "key" : 0,
        "count" : 18565
      }, {
        "key" : 1000,
        "count" : 33
      }, {
        "key" : 2000,
        "count" : 14
      }, {
        "key" : 3000,
        "count" : 5
      },
```

```
        (...)
      ]
    }
  }
}
```

可以看到，我们在0~1000范围内有18 565个文档，第二个1000~2000范围内有33个文档，以此类推。

date_histogram切面

对于数值型字段，除了histogram切面类型外，Elasticsearch还提供了date_histogram切面类型，用于日期型字段。date_histogram类型允许我们使用诸如year、month、week、day、hour或minute等常量作为interval属性的值。例如，可以发送如下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "date_histogram_test" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "day"
      }
    }
  }
}
```

> 在数值型和date_histogram切面中，都可以使用key_field、key_value、key_script和value_script等属性，这在前面说明terms切面时讨论过。

6.2.8 数值型字段统计数据的计算

statistical切面允许我们为数值型字段计算统计数据。在返回数据中，我们得到计数、总和、平方和、平均值、最小值、最大值、方差和标准偏差等统计信息。例如，为total字段计算统计，可以执行如下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "statistical_test" : {
      "statistical" : {
        "field" : "total"
      }
    }
  }
}
```

在结果中，得到如下输出：

```
{
  (...)
  "facets" : {
    "statistical_test" : {
      "_type" : "statistical",
      "count" : 18643,
      "total" : 645706.0,
      "min" : 0.0,
      "max" : 57676.0,
      "mean" : 34.63530547658639,
      "sum_of_squares" : 1.2490405256E10,
      "variance" : 668778.6853747752,
      "std_deviation" : 817.7889002516329
    }
  }
}
```

前面的输出返回了以下统计信息。

- _type：定义了切面的类型。
- count：定义了在给定字段上有值的文档的数目。
- total：定义了给定字段所有值的总和。
- min：定义了字段的最小值。
- max：定义了字段的最大值。
- mean：定义了给定字段的均值。
- sum_of_squares：定义了给定字段上值的平方和。
- variance：定义了给定字段值的方差。
- std_deviation：定义了给定字段值的标准偏差。

> 注意，和terms切面中一样，也可以在statistical切面中使用script和fields属性。

6.2.9　词条统计数据的计算

除terms和statistical切面外，Elasticsearch允许使用terms_stats切面。它结合了statistical和terms切面类型，提供给我们基于一个字段的值来统计另一个字段的能力。如果想要total字段上的切面，又想在tags字段基础上划分这些值，运行以下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "total_tags_terms_stats" : {
      "terms_stats" : {
        "key_field" : "tags",
```

```
        "value_field" : "total"
      }
    }
  }
}
```

我们指定了key_field属性，其中包含了提供词条的字段的名称；value_field属性，包含数值字段的名称。以下是我们从Elasticsearch得到的部分结果：

```
{
  (...)
  "facets" : {
    "total_tags_terms_stats" : {
      "_type" : "terms_stats",
      "missing" : 54715,
      "terms" : [ {
        "term" : "personal",
        "count" : 1063,
        "total_count" : 254,
        "min" : 0.0,
        "max" : 322.0,
        "total" : 707.0,
        "mean" : 2.783464566929134
      }, {
        "term" : "me",
        "count" : 715,
        "total_count" : 218,
        "min" : 0.0,
        "max" : 138.0,
        "total" : 710.0,
        "mean" : 3.256880733944954
      }
      (...)
      ]
    }
  }
}
```


可以看到，切面结果根据词条进行划分。注意每个词条都返回了一组统计，跟ranges切面中一样（这些值的含义参见6.2.6节）。这是因为我们使用了数值型字段（total字段）来为每个字段计算切面的值。

6.2.10 地理切面

我们想讨论的最后一个切面计算类型是geo_distance切面。它能够获得落入到从给定位置到某距离范围内的文档的数目信息。假设我们在索引中的文档有个location字段，存储了地理位置。现在想象一下我们要获取距离某一特定点的有关文档，比如从10.0、10.0这个点，想知道有多少文档落在距离这个点10公里以内，多少落在从10到100公里，又有多少落在100公里以外。为此，运行以下查询（6.6节将讲述如何定义location字段）：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "spatial_test" : {
      "geo_distance" : {
         "location" : {
          "lat" : 10.0,
          "lon" : 10.0
        },
        "ranges" : [
          { "to" : 10 },
          { "from" : 10, "to" : 100 },
          { "from" : 100 }
        ]
      }
    }
  }
}
```

在前面的查询中，我们定义了计算距离的基准点的纬度（lat属性）和经度（lon属性）。注意，我们传入lat和lon属性的对象的名字。该对象的名字必须等于字段的名字。第二件事是ranges数组，每个范围可以使用to、from或两者来定义。

除了上述属性外，还可以设置unit属性（默认情况下，km表示公里，mi表示英里）和distance_type属性（默认情况下，arc精度更高，plane执行得更快）。

6.2.11　切面结果的过滤

查询中包含的过滤器并不会缩小切面结果的范围，切面的计算时基于与你的查询匹配的文档。然而，你可能希望在你的切面定义中包含你想要的过滤器。基本上，在3.5节中讨论的任意一个过滤器都可以用在切面中，你所需做的只是在切面名称下包含一个额外的节点。

如果我们希望查询所有文档，但只对tags字段中含有fashion词条的文档来为多值字段tags计算切面，可以执行以下查询：

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "tags" : {
      "terms" : { "field" : "tags" },
      "facet_filter" : {
        "term" : { "tags" : "fashion" }
      }
    }
  }
}
```

可以看到，在切面类型（在上面查询中的terms）同一水平上，添加了一个额外的facet_

filter节点。你只需要记住facet_filter部分采用与2.2节描述的过滤器相同的逻辑来构建。

6.2.12 内存考虑

切面可能对内存有要求，尤其是在索引中有大量文档和很多独特值的情况下。内存要求高是因为，Elasticsearch需要把数据加载到字段数据缓存中来计算切面的值。随着doc值的引入（2.2节讨论过），Elasticsearch可以使用这个数据结构来执行所有需要字段数据缓存的操作，比如切面和排序。在大量数据的情况下，使用doc值是明智的。老的方法同样有用，比如使用更低精度的日期来降低字段的基（cardinality），不分析字符串字段，或者尽量使用short、integer或float而不是long和double。如果这些都没用，你可能需要给Elasticsearch配置更高的堆内存，甚至增加更多服务器，把索引分到更多分片上。

6.3 使用建议器

从Elasticsearch 0.90起，可以使用所谓的建议器（suggester）。我们可以把建议器看成这样的功能：在考虑性能的情况下，允许纠正用户的拼写错误，以及构建一个自动完成功能。本节将带你走进建议器的世界，但是，这不是一个全面的指导。描述建议器的所有细节将是非常广泛的话题，超出了本书的范围。如果你想了解更多关于建议器的内容，请参阅Elasticsearch官方文档（http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-suggesters.html）或我们的书，*Mastering ElasticSearch*，Packt出版。

6.3.1 可用的建议器类型

Elasticsearch提供了以下三种建议器的类型。

- term：这种建议器更正每个传入的单词，在非短语查询中很有用，比如单词条查询。
- phrase：这种建议器工作在短语上，返回一个恰当的短语。
- completion：这种建议器旨在提供快速高效的自动完成结果。

我们将分别讨论这些建议器。此外，也可以使用_suggest这个REST端点。

6.3.2 包含建议器

现在，让我们试试在查询结果中得到建议。例如，使用match_all查询并尝试为serlock holnes短语得到一个建议，该短语包含两个拼写错误的词条。为此，我们将执行以下命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
```

```
    },
    "suggest" : {
      "first_suggestion" : {
        "text" : "serlock holnes",
        "term" : {
        "field" : "_all"
      }
    }
  }
}'
```

如果希望为同样的文本得到多个建议，可以把建议嵌入到suggest对象中，并把text属性设置为选择的建议对象。如果想得到title和_all字段上serlock holne文本的建议，执行以下命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "text" : "serlock holnes",
    "first_suggestion" : {
      "term" : {
        "field" : "_all"
      }
    },
    "second_suggestion" : {
      "term" : {
        "field" : "title"
      }
    }
  }
}'
```

建议器的响应

现在来看看我们发送的第一个查询的响应。你可以猜到，响应将包括查询结果和建议：

```
{
  "took" : 1,
  "timed_out" : false,
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [
    ...
    ]
  },
  "suggest" : {
    "first_suggestion" : [ {
      "text" : "serlock",
```

```
          "offset" : 0,
          "length" : 7,
          "options" : [ {
            "text" : "sherlock",
            "score" : 0.85714287,
            "freq" : 1
          } ]
        }, {
          "text" : "holnes",
          "offset" : 8,
          "length" : 6,
          "options" : [ {
            "text" : "holmes",
            "score" : 0.8333333,
            "freq" : 1
          } ]
        } ]
      }
    }
```

可以看到，在响应中得到了搜索结果和建议（省略了查询的响应，以便这个例子更容易阅读）。

term建议器为text参数中的每个词条返回了一个可能的建议列表。每个词条，term建议器都将返回一组可能的建议。看看为serlock词条返回的数据，可以看到原始的单词（text参数）、它在原始text参数中的偏移量（offset参数）和它的长度（length参数）。

options数组包含了给定单词的建议，如果Elasticsearch没有找到任何建议，它将为空。数组里的每个条目都是一个建议，通过下面的属性来描述。

- text：该属性定义了建议的文本。
- score：该属性定义了建议的得分，得分越高，建议越好。
- freq：该属性定义了建议的频率。频率代表在我们执行建议查询的索引上，该单词出现在文档中的次数。

6

6.3.3 term建议器

term建议器基于字符串编辑距离（string edit distance）工作。这意味着，通过更少字符的更改、添加或删除就可以让建议单词和原始单词一样，那么这样的建议最好。例如，我们来看单词worl和work。为了把worl改成work，需要把l改成k，这意味着编辑距离为1。当然，提供给建议器的文本是经过分析的，因此它将选择词条来建议。

1. term建议器的配置选项

term建议器中最常见和常用的配置项同样可以用在基于term建议器实现的所有建议器上。目前，有phrase建议器，当然还有基本的term建议器。可用的选项如下。

- text：这个选项定义了我们希望得到建议的文本。此参数是必须的。
- field：这是另一个必须提供的参数。field参数设置了为哪个字段生成建议。
- analyzer：这个选项定义了分析器的名字，该分析器用作分析text参数提供的文本。如果未设置，Elasticsearch将使用field参数所指定的字段所用的分析器。
- size：这个参数默认为5，指定了text参数中每个词条可以返回的建议的最大数字。
- sort：此选项允许指定Elasticsearch返回的建议如何排序。默认情况下，此选项设置成score，Elasticsearch将首先按照建议的得分排，然后按文档频率，最后按词条排。第二个可能值为frequency，意味着结果首先按文档频率排，然后按分数，最后按词条。

2. 额外的term建议器选项

除了前面提到的常见term建议器选项外，Elasticsearch还提供了专对term建议器有意义的一些额外选项，其中的一些选项如下。

- lowercase_terms：当设置为true时，此选项将告诉Elasticsearch把从text字段生成的所有经分析后的词条变成小写。
- max_edits：此选项的默认值为2，指定建议词条允许的最大编辑距离。Elasticsearch允许它设置为1或2。
- prefix_len：此选项默认为1。如果我们在建议器的性能上挣扎，增加这个值可以提高整体性能，因为这时需要处理更少数量的建议。
- min_word_len：此选项默认为4。指定了返回的建议列表中词条的最少字符数。
- shard_size：此选项默认值为size参数的值，可以用它设定每个分片上应该读取的最大建议数量。把该参数设置为比size参数大的值，会得到更准确的文档频率，但代价是建议器性能的降低。

6.3.4　phrase建议器

term建议器提供了一个很好的方式基于每个词条来纠正用户的拼写错误，但对短语来说不够好。这就是引入phrase建议器的原因。它建立在term建议器之上，但添加了额外的短语计算逻辑。

我们以一个示例开始介绍如何使用phrase建议器。这次我们将省略查询中的query部分，为此执行下面命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "suggest" : {
    "text" : "sherlock holnes",
    "our_suggestion" : {
      "phrase" : { "field" : "_all" }
    }
  }
}'
```

可以看到上面的命令，几乎跟使用term建议器时一模一样。只是，指定了phrase类型，而不是term类型。上述命令的响应将如下所示：

```
{
  "took" : 1,
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [
    ...
    ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "sherlock holnes",
      "offset" : 0,
      "length" : 15,
      "options" : [ {
        "text" : "sherlock holmes",
        "score" : 0.12227806
      } ]
    } ]
  }
}
```

正如你所看到的，返回的响应与一个term建议器返回的非常类似，只是不再返回一个词，而是一个结合的短语。

配置

因为phrase建议器基于term建议器，因此它可以使用term建议器提供的一些配置选项，它们是text、size、analyzer和shard_size。除了这些属性外，phrase建议器公开了如下一些额外选项。

- max_errors：此选项指定了可纠正的短语中包含错误词条的最大数目、或最大百分比。该参数可以设置为一个整型值，比如1，也可以设成0到1之间的一个浮点数，这时它被当成百分比。默认情况下，该值为1。意味着给定短语中最多包含1个错误拼写的词条。
- separator：此选项默认是空白符，指定了用来在结果字段中分割词条的分隔符。

6.3.5 completion建议器

completion建议器允许我们创建自动完成功能，并且性能很好，这是因为你可以在索引中存储复杂结构，而不是在查询时计算。

为了使用这种基于前缀的建议器，我们需要正确建立索引数据，在其中包含一个叫

completion的专用字段。为了说明如何使用这种建议器，假设建立一个自动完成功能来显示图书的作者。除了作者的名字以外，我们希望返回该作者写的图书的标识符。先用下面的命令创建一个authors索引：

```
curl -XPOST 'localhost:9200/authors' -d '{
  "mappings" : {
    "author" : {
      "properties" : {
        "name" : { "type" : "string" },
        "ac" : {
          "type" : "completion",
          "index_analyzer" : "simple",
          "search_analyzer" : "simple",
          "payloads" : true
        }
      }
    }
  }
}'
```

我们的索引将包含一个单一类型，叫author。每个文档有两个字段：name字段和ac字段，它们将用来构建自动完成。我们使用completion类型定义ac字段，另外在索引和查询时都使用simple分析器。最后一个是payload：随建议一起返回一个额外的选项信息，在我们的例子中，它将是图书标识符的数组。

1. 索引数据

为了创建索引，除了我们通常提供的信息以外，还需要提供一些额外的信息。看看下面的命令，它索引了两个描述作者的文档：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : {
    "input" : [ "fyodor", "dostoevsky" ],
    "output" : "Fyodor Dostoevsky",
    "payload" : { "books" : [ "123456", "123457" ] }
  }
}'
curl -XPOST 'localhost:9200/authors/author/2' -d '{
  "name" : "Joseph Conrad",
  "ac" : {
    "input" : [ "joseph", "conrad" ],
    "output" : "Joseph Conrad",
    "payload" : { "books" : [ "121211" ] }
  }
}'
```

注意ac字段的数据结构。我们提供了input、output和payload属性。可选的payload属性用于提供额外的返回信息。input属性提供了建议器用来生成自动完成功能的输入信息，它将

被用于匹配用户的输入。可选的output属性告诉建议器，应该为文档返回什么数据。

也可以省略额外的参数部分，使用我们习惯的方式来建立索引，就像下面的例子：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : "Fyodor Dostoevsky"
}'
```

然而，因为completion建议器在底层使用了FST，因此如果我们使用ac字段的第二部分来查找，将找不到前面的文档。这就是为什么我们认为使用第一种建立索引的方式更加方便，因为可以控制想要匹配什么，以及想要显示什么。

2. 查询索引中的completion建议器数据

如果我们想找到作者名字以fyo开头的文档，可以执行下面的命令：

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
  "authorsAutocomplete" : {
    "text" : "fyo",
    "completion" : {
      "field" : "ac"
    }
  }
}'
```

在看结果之前，让我们讨论一下这个查询。你可以看到，我们在_suggest端点执行命令，因为我们不想执行标准查询，只对自动完成结果感兴趣。查询很简单；我们将其名称设置为authorsAutocomplete，设置想要自动完成的文本（text属性），并且添加包含配置的completion对象。上述命令的结果将如下所示：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

在响应中你可以看到，我们得到了要找的文档，并包含payload信息。

6

也可以使用模糊搜索来容忍拼写错误。为此，在查询中包含一个额外的fuzzy节点。例如，要启用completion建议器的模糊匹配，并设置最大编辑距离为2（这意味着最多允许2个错误），可以发送以下查询：

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
  "authorsAutocomplete" : {
    "text" : "fio",
    "completion" : {
      "field" : "ac",
      "fuzzy" : {
        "edit_distance" : 2
      }
    }
  }
}'
```

尽管犯了一个拼写错误，我们仍将获得跟以前一样的结果。

3. 定制权重

默认情况下，词频将用来决定前缀建议器返回文档的权重。然而，这有时不是最好的方案。这时，通过为定义成completion的字段指定一个weight属性来定义建议的权重，是很有用的。weight属性应该设置成一个整型值，值越高，建议越重要。如果想为示例中的第一个文档指定一个权重，执行如下命令：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : {
    "input" : [ "fyodor", "dostoevsky" ],
    "output" : "Fyodor Dostoevsky",
    "payload" : { "books" : [ "123456", "123457" ] },
    "weight" : 30
  }
}'
```

现在，执行示例查询，结果将如下所示：

```
{
  ...
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 30.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

看看结果中score的变化，之前的例子中它是1.0，现在是30.0。因为我们在索引时把weight参数设成了30。

6.4 预匹配器

你有没有想过如果我们使用反传统模式的查询来查找文档会如何？查找与文档匹配的查询有意义吗？在很多解决方案中，此模型都很有用。当你对一个无限的输入数据流进行操作并搜索特定事件的出现时，可以使用此方法。这可以用于检测监控系统中的故障，或开发这样的功能："商店中有符合预定义条件的商品出现时，告诉我"。本节将介绍Elasticsearch预匹配器的工作原理，以及它如何处理最后这个例子。

6.4.1 示例索引

所有关于预匹配器的例子，都将使用一个叫notifier的索引，它使用下面的命令来创建：

```
curl -XPOST 'localhost:9200/notifier' -d '{
  "mappings": {
    "book" : {
      "properties" : {
        "available" : {
          "type" : "boolean"
        }
      }
    }
  }
}'
```

我们只定义了一个字段。其他字段将使用Elasticsearch的无模式特性，它们的类型将被猜测。

6.4.2 预匹配器的准备

一个预匹配器看上去像Elasticsearch中的一个额外的文档类型。这意味着我们可以存储任何文档，然后像搜索任何索引中的普通类型一样搜索它们。然而，预匹配器允许我们倒转逻辑：对查询建立索引，然后发送文档给Elasticsearch，看看索引中哪个索引被匹配到。以第2章中的library为例，试着把查询索引到预匹配器中。假设任何与预定义的条件匹配的图书变得可用时，通知我们的用户。

看看下面的query1.json文件，包含了用户生成的一个示例查询：

```
{
  "query" : {
    "bool" : {
      "must" : {
```

```
          "term" : {
            "title" : "crime"
          }
        },
        "should" : {
          "range" : {
            "year" : {
              "gt" : 1900,
              "lt" : 2000
            }
          }
        },
        "must_not" : {
          "term" : {
            "otitle" : "nothing"
          }
        }
      }
    }
  }
```

此外用户还可以使用我们的假想用户界面来定义过滤器。为了说明这些功能，我们得到了一个用户查询，query2.json文件中的这个查询应该找到所有写于2010年之前的可用图书。查询如下所示：

```
{
  "query" : {
    "filtered": {
      "query" : {
        "range" : {
          "year" : {
            "lt" : 2010
          }
        }
      },
      "filter" : {
        "term" : {
          "available" : true
        }
      }
    }
  }
}
```

现在，把这两个查询都注册到预匹配器中（注意只是注册查询，还没有索引任何文档）。为此，执行下面的命令：

```
curl -XPUT 'localhost:9200/notifier/.percolator/1' -d @query1.json
curl -XPUT 'localhost:9200/notifier/.percolator/old_books' -d
  @query2.json
```

在前面的示例中，使用两个完全不同的标识符。这样做是为了表明我们可以使用一个最好的

描述查询的标识符。想把查询注册成什么名字完全取决于我们。

现在可以使用预匹配器了。我们的应用程序向预匹配器提供文档，并检查Elasticsearch是否找到相应的查询。这正是预匹配器允许我们做的：反转搜索逻辑。不再是索引文档并对其执行搜索，而是存储查询并发送文档。作为结果，Elasticsearch讲告诉我们哪些查询与当前文档匹配。

我们用一个与上面两个查询都匹配的示例文档：它有所需的标题、发布日期，并注明当前是否可用。发送这个文档的命令如下所示：

```
curl -XGET 'localhost:9200/notifier/book/_percolate?pretty' -d '{
  "doc" : {
    "title": "Crime and Punishment",
    "otitle": "ПреступлéнХие и наказáнХие",
    "author": "Fyodor Dostoevsky",
    "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"],
    "tags": [],
    "copies": 0,
    "available" : true
  }
}'
```

正如我们所料，Elasticsearch响应中包含匹配查询的标识符。这种响应将如下所示：

```
"matches" : [ {
  "_index" : "notifier",
  "_id" : "1"
}, {
  "_index" : "notifier",
  "_id" : "old_books"
} ]
```

这很美妙。请注意该查询使用的端点：`_percolate`。索引的名字对应到存储查询的索引，类型等于映射中定义的类型。

> 响应格式中包含索引和查询标识符的信息。当我们搜索多个索引时，可以包含这些信息。如果只是搜索一个索引，添加一个额外的查询参数`percolate_format=ids`，响应将变成：
>
> **"matches" : ["3"]**。

6.4.3 深入

因为注册在预匹配器中的查询实际上也是文档，可以向Elasticsearch发送一个正常查询，来选择在预匹配过程中应该使用存储在`.percolator`索引中的哪个查询。这听上去有点奇怪，但确实提供了很多可能性。我们的图书馆有几组用户，假设有些用户有权限借阅一些非常罕见的书，

或者我们在城市中有几个分馆，用户可以申报他们想去哪个分馆借阅图书。

看看如何使用预匹配器来实现这样的用例。为此，需要更新映射。使用下面的命令来添加一个.percolator类型：

```
curl -XPOST 'localhost:9200/notifier/.percolator/_mapping' -d '{
  ".percolator" : {
    "properties" : {
      "branches" : {
        "type" : "string",
        "index" : "not_analyzed"
      }
    }
  }
}'
```

现在，为了注册这个查询，使用下面的命令：

```
curl -XPUT 'localhost:9200/notifier/.percolator/3' -d '{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "branches" : ["brA", "brB", "brD"]
}'
```

上面的例子中，用户对任何标题中含有crime词条的图书感兴趣（词条查询负责这个），想从三个所列分馆中借阅这些书。当指定映射时，我们定义了branches字段是未经分析的字符串字段。现在可以在之前发送的文档中包含一个查询，看下我们如何做到这一点。

我们的图书系统得到了这本书，准备报告这本书并且检查是否有人对该书感兴趣。为了检查这个，发送一个描述该书的文档，并且在请求中添加一个额外查询，该查询将会限制用户为对brB分馆感兴趣的用户。这样的请求如下所示：

```
curl -XGET 'localhost:9200/notifier/book/_percolate?pretty' -d '{
  "doc" : {
    "title": "Crime and Punishment",
    "otitle": "Преступлéние и наказáние",
    "author": "Fyodor Dostoevsky",
    "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"],
    "tags": [],
    "copies": 0,
    "available" : true
  },
  "size" : 10,
  "query" : {
    "term" : {
      "branches" : "brB"
    }
```

```
  }
}'
```

如果一切都正确执行，Elasticsearch返回的响应应该如下（我们索引的查询以3作为标识符）：

```
"total" : 1,
"matches" : [ {
  "_index" : "notifier",
  "_id" : "3"
} ]
```

另外有一点要注意的是size参数。它允许我们限制返回匹配项的数目。这对于安全是必需的：你应该知道你在做什么，因为返回的匹配查询数目可能非常大，这可能意味着内存相关的问题。

如果我们可以在发往预匹配器的文档中使用查询，为什么不能使用Elasticsearch的其他功能呢？当然可以，例如，下面发送的文档包含了一个聚合：

```
{
  "doc": {
    "title": "Crime and Punishment",
    "available": true
  },
  "aggs" : {
    "test" : {
      "terms" : {
        "field" : "branches"
      }
    }
  }
}
```

可以包含查询、过滤器、切面和聚合。那么高亮呢？请看下面的示例文档：

```
{
  "doc": {
    "title": "Crime and Punishment",
    "year": 1886,
    "available": true
  },
  "size" : 10,
  "highlight": {
    "fields": {
      "title": {}
    }
  }
}
```

可以看到，它包含了一个高亮节点。响应的一个片段将如下所示：

```
{
  "_index": "notifier",
  "_id": "3",
  "highlight": {
```

```
    "title": [
      "<em>Crime</em> and Punishment"
    ]
  }
}
```

> 注意，预匹配器功能支持的查询类型有些限制。在目前的实现中，父子查询和嵌套查询不可用，所以你不能使用has_child、top_children、has_parent和nested等查询。

1. 得到匹配查询的数量

有时，你不关心匹配到的查询本书，你所需要的只是匹配查询的数量。在这种情况下，发送文档到标准percolator端点的效率不高。Elasticsearch公开了_percolate/count端点来高效地处理这种情况。这种命令的一个例子如下：

```
curl -XGET 'localhost:9200/notifier/book/_percolate/count?pretty' -d '{
 "doc" : { ... }
 }'
```

2. 索引文档的预匹配

还有一种可能，如果我们想检查哪些查询跟已经编制到索引的文档匹配，怎么办？当然，这是可以做到的。看看下面的命令：

```
curl -XGET 'localhost:9200/library/book/1/_percolate?percolate_
index=notifier'
```

此命令针对percolator_index参数指定的预匹配器索引，来检查library索引中标识符为1的文档。请记住，默认情况下，Elasticsearch使用跟文档同样索引中的预匹配器，这就是我们指定percolator_index参数的原因。

6.5　文件的处理

下一个我们要讨论的用例是搜索文件的内容。最显而易见的方法是在应用中添加一段逻辑，负责读取文件，从中提取有价值的信息，构建成JSON对象，最后把它们构建到Elasticsearch的索引中。

当然，这个方法是可行的，你可以这样做。但我们想向你展现另一张方法。可以把文档发送给Elasticsearch，让它来做内容提取和索引建立。这需要安装一个额外的插件。注意，第7章将描述插件，所以在此我们略过详细的描述，执行下面的命令来安装attachments插件：

```
bin/plugin -install elasticsearch/elasticsearch-mapper-
  attachments/2.0.0.RC1
```

重启Elasticsearch后，它将奇迹般地获取一个新技能，我们现在来用用看。先准备一个新索

引，映射如下：

```
{
  "mappings" : {
    "file" : {
      "properties" : {
        "note" : { "type" : "string", "store" : "yes" },
        "book" : {
          "type" : "attachment",
          "fields" : {
            "file" : { "store" : "yes", "index" : "analyzed" },
            "date" : { "store" : "yes" },
            "author" : { "store" : "yes" },
            "keywords" : { "store" : "yes" },
            "content_type" : { "store" : "yes" },
            "title" : { "store" : "yes" }
          }
        }
      }
    }
  }
}
```

可以看到，我们有一个book类型，用来存储文件的内容，还定义了一些嵌套字段，如下所示。

- file：该字段定义了文件的内容。
- date：该字段定义了文件的创建日期。
- author：该字段定义了文件的作者。
- keywords：该字段定义了连接文档的一些额外关键字。
- content_type：该字段定义了文档的mime类型。
- title：该字段定义了文档的标题。

如果存在，这些字段将从文件中提取。在我们的例子中，把这些字段标识为stored，这可以在搜索结果中看到它们的值。此外，我们定义了note字段，这是一个普通字段，它将被我们而不是被插件使用。

6

现在，来准备文档。看看存储在index.json文件中的示例文档：

```
{
  "book" : "UEsDBBQABgAIAAAAIQDpURCwjQEAAMIFAAATAAgCW0NvbnR
    lbnRfVHlwZXNdLnhtbCCiBAIooAA…",
  "note" : "just a note"
}
```

你可以看到，book字段中有一些奇怪的内容。它是使用Base64算法编码的文件内容（注意这只是它的一小部分，为清楚起见，我们省略了其他部分）。因为文件内容可以是二进制的，不能简单地包含进JSON对象中，因此Elasticsearch的作者要求我们使用该算法来编码文件内容。在Linux操作系统上，有一个简单命令用来把文档内容编码成Base64，比如，可以使用下列命令：

```
base64 -i example.docx -o example.docx.base64
```

假设你已经成功创建了一个合法的Base64版本的文档。现在，可以通过执行下面命令来索引该文档：

```
curl -XPUT 'localhost:9200/media/file/1?pretty' -d @index.json
```

这很简单。Elasticsearch在后台解码文件，提取内容，并在我们的索引中创建适当的条目。

现在，创建用于查找文档的查询（放置到query.json文件中），如下所示：

```
{
  "fields" : ["title", "author", "date", "keywords",
"content_type", "note"],
  "query" : {
    "term" : { "book" : "example" }
  }
}
```

如果你仔细看过前面几章，上面的查询应该很容易理解。我们请求book字段的example一词。我们编码过的示例文档，包含以下文本：This is an example document for 'Elasticsearch Server' book。因此，我们刚才的示例查询应该与我们的文档匹配。执行以下命令来验证这个假设：

```
curl -XGET 'localhost:9200/media/_search?pretty' -d @query.json
```

如果一切顺利，应该得到类似下面的响应：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.095891505,
    "hits" : [ {
      "_index" : "media",
      "_type" : "file",
      "_id" : "1",
      "_score" : 0.095891505,
      "fields" : {
        "book.date" : [ "2014-02-08T09:34:00.000Z" ],
        "book.content_type" : [ "application/vnd.openxmlformats-
          officedocument.wordprocessingml.document" ],
        "note" : [ "just a note" ],
        "book.author" : [ "Rafał Kux, Marek Rogozizski" ]
      }
    } ]
```

```
    }
  }
```

看下结果，你可以发现内容类型是application/vnd.openxmlformats-officedocument.wordpro-cessingml.document。你可以猜到我们的文档使用Microsoft Office创建的，扩展名很可能是docx。我们还看到了从文档中提取的其他啊额外字段，比如作者、修改日期。一切正常！

添加文件的额外信息

索引文件时，一个显而易见的需求是在结果列表中返回文件名。当然，我们可以在文档中添加一个文件名的字段，但Elasticsearch允许在文件对象中存储这些信息。我们只需在发送给Elasticsearch的文档中添加_name字段。如果想把example.docx这个名字索引成文档的名字，可以发送如下文档：

```
{
  "book" :
    "UEsDBBQABgAIAAAAIQDpURCwjQEAAMIFAAATAAgCW0NvbnRlbnRfVHl
    wZXNdLnhtbCCiBAIooAA…",
  "_name" : "example.docx",
  "note" : "just a note"
}
```

通过包含_name字段，Elasticsearch将在结果列表包括名称。文件名将作为_source字段的一部分。但是，如果你使用fields属性，又要在结果中返回文件名称，就应该添加_source字段为此属性中的一个条目。

最后，你可以使用content_type字段来存储mime类型信息，正如我们使用_name字段来存储文件名。

6.6 地理

通常我们是从全文搜索的角度来看待像Elasticsearch这样的搜索服务器的。但这只是全局的一部分。有时，全文搜索是不够的。想象一个本地服务的搜索，对最终用户来说，最重要的是搜索的准确性。准确性不仅指全文搜索中正确的结果，还包括结果跟某位置越近越好。在一些情况下，这意味着对地理名称的文本搜索，比如城市或街道；但在另外一些情况下，我们发现能够给予索引文档的地理坐标来搜索是非常有用的。而这个也是Elasticsearch能够处理的功能。

6.6.1 为空间搜索准备映射

为了探讨空间搜索功能，让我们准备一个包含城市列表的索引。这将是一个非常简单的索引，包含一个名为poi的类型（poi：point of interest，兴趣点），城市的名称和它的坐标。映射如下：

```
{
  "mappings" : {
    "poi" : {
      "properties" : {
        "name" : { "type" : "string" },
        "location" : { "type" : "geo_point" }
      }
    }
  }
}
```

假设我们把这个定义存到mapping.json文件中，可以通过执行下面的命令来创建一个索引：

```
curl -XPUT localhost:9200/map -d @mapping.json
```

唯一的新东西是geo_point类型，用在location字段上。通过它，我们可以存储城市的地理位置。

6.6.2　示例数据

我们文档的示例文件如下所示：

```
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 1 }}
{ "name" : "New York", "location" : "40.664167, -73.938611" }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 2 }}
{ "name" : "London", "location" : [-0.1275, 51.507222] }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 3 }}
{ "name" : "Moscow", "location" : { "lat" : 55.75, "lon" : 37.616667
}}
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 4 }}
{ "name" : "Sydney", "location" : "-33.859972, 151.211111" }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 5 }}
{ "name" : "Lisbon", "location" : "eycs0p8ukc7v" }
```

为了执行批量请求，我们添加了有关索引名称，类型和文档的唯一标识符。所以，我们现在可以使用以下命令轻松导入此数据：

```
curl -XPOST http://localhost:9200/_bulk --data-binary @documents.json
```

应该仔细看看location字段。我们可以在坐标中使用各种符号，使用字符串、数字对或者一个对象来提供经纬度。注意使用字符串和数组来提供地理位置时，在经度和纬度参数上有不同的顺序。最后一条记录展现了另外一种提供坐标的办法，即使用地理散列值（geohash value），这里可以找到该符号的描述：http://en.wikipedia.org/wiki/Geohash。

6.6.3　示例查询

现在，让我们看看几个例子，说明如何使用坐标，以及如何解决在现代应用中，在全文搜索

中搜索地理数据的常见需求。

1. 基于距离的排序

先来看一个非常常见的需求：按照与给定地点的距离来对结果排序。在我们的例子中，希望得到所有的城市，并按照距法国首都巴黎的距离来排序。为此，发送以下查询到Elasticsearch中：

```
{
  "query" : {
    "match_all" : {}
  },
  "sort" : [{
    "_geo_distance" : {
      "location" : "48.8567, 2.3508",
      "unit" : "km"
    }
  }]
}
```

如果你记得3.8节，就会注意到格式有细微不同。我们使用_geo_distance键来表明按照距离进排序。我们必须提供基准地点（location属性，在我们的例子中，包含了巴黎的信息），以及指定用在结果中的单位，可用的值包含km和mi，分别表示公里和英里。该查询的结果如下所示：

```
{
  "took" : 102,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 5,
    "max_score" : null,
    "hits" : [ {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "2",
      "_score" : null, "_source" : { "name" : "London", "location" : [-
        0.1275, 51.507222] },
      "sort" : [ 343.46748684411773 ]
    }, {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "5",
      "_score" : null, "_source" : { "name" : "Lisbon", "location" :
        "eycs0p8ukc7v" },
      "sort" : [ 1453.6450747751787 ]
    }, {
      "_index" : "map",
      "_type" : "poi",
```

```
        "_id" : "3",
        "_score" : null, "_source" : { "name" : "Moscow", "location" :
          { "lat" : 55.75, "lon" : 37.616667 }},
        "sort" : [ 2486.2560754763977 ]
      }, {
        "_index" : "map",
        "_type" : "poi",
        "_id" : "1",
        "_score" : null, "_source" : { "name" : "New York", "location"
          : "40.664167, -73.938611" },
        "sort" : [ 5835.763890418129 ]
      }, {
        "_index" : "map",
        "_type" : "poi",
        "_id" : "4",
        "_score" : null, "_source" : { "name" : "Sydney", "location" :
          "-33.859972, 151.211111" },
        "sort" : [ 16960.04911335322 ]
      } ]
    }
  }
```

至于排序的其他例子，Elasticsearch显示了用于排序的值的信息。让我们看看突出显示的记录。可以看到，巴黎和伦敦之间的距离约343公里，你可以看到地图上确实如此。

2. 边界框过滤

我们要展现的下一个例子是，缩小结果到一个被矩形框住的选定区域。当我们需要在地图上显示结果，或者允许用户标记地图区域来搜索时，这是非常方便的。

3.5节介绍过过滤器，但当时我们没有提到空间过滤器。下面的查询展示了如何使用边界框来过滤：

```
{
  "filter" : {
    "geo_bounding_box" : {
      "location" : {
        "top_left" : "52.4796, -1.903",
        "bottom_right" : "48.8567, 2.3508"
      }
    }
  }
}
```

在前面的示例中，我们通过提供左上和右下坐标，选择了伯明翰和巴黎之间的一个地图片段。这两个坐标足以指定任何我们想要的矩形区域，Elasticsearch会做剩下的计算。下面的屏幕快照显示指定的地图上的矩形：



可以看到，我们的数据中唯一符合条件的城市是伦敦。运行上述查询来检查Elasticsearch是否知道这个。看看返回的结果，如下所示：

```
{
  "took" : 9,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "2",
      "_score" : 1.0, "_source" : { "name" : "London", "location" :
        [-0.1275, 51.507222] }
    } ]
  }
}
```

可以看到，Elasticsearch又一次跟地图一致。

3. 距离的限制

最后一个示例显示下一个常见需求：把结果限定为离基准点一个选定的距离之内。如果我们把结果限定为离巴黎半径500公里以内，可以使用下面的过滤器：

```
{
  "filter" : {
    "geo_distance" : {
      "location" : "48.8567, 2.3508",
        "distance" : "500km"
    }
  }
}
```

如果一切正常，Elasticsearch应该只为上述查询返回一条记录，该记录应该是London。请读者自行验证。

6.6.4　任意地理形状

有时，使用单一的地理点或矩形是不够的。在这种情况下，需要更复杂的东西，Elasticsearch提供了定义形状的功能来解决这个需求。为了展示我们在Elasticsearch中如何利用自定义形状限制，需要修改索引，引入geo_shape类型。我们的新映射如下所示（将用它来创建名为map2的索引）：

```
{
  "poi" : {
    "properties" : {
      "name" : { "type" : "string", "index": "not_analyzed" },
      "location" : { "type" : "geo_shape" }
    }
  }
}
```

接下来，修改示例数据，以匹配新的索引结构，如下所示：

```
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 1 }}
{ "name" : "New York", "location" : { "type": "point", "coordinates":
[-73.938611, 40.664167] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 2 }}
{ "name" : "London", "location" : { "type": "point", "coordinates":
[-0.1275, 51.507222] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 3 }}
{ "name" : "Moscow", "location" : { "type": "point", "coordinates": [
37.616667, 55.75]}}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 4 }}
{ "name" : "Sydney", "location" : { "type": "point", "coordinates":
[151.211111, -33.865143]}}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 5 }}
{ "name" : "Lisbon", "location" : { "type": "point", "coordinates":
[-9.142685, 38.736946] }}
```

geo_shape类型字段的结构不同于geo_point。它的语法称为GeoJSON（http://en.wikipedia.org/wiki/GeoJSON）。它允许我们定义各种地理类型。总结一下可以使用在查询期间的类型，至少包括我们认为最有用的那些。

1. 点

一个点由一个表定义，第一个元素是经度，第二个元素是纬度。该形状的一个例子如下所示：

```
{
  "location" : {
    "type": "point",
    "coordinates": [-0.1275, 51.507222]
  }
}
```

2. 包络线

一个包络线（Envelope）通过提供左上和右下两个坐标定义一个框。例子如下所示：

```
{
  "type": "envelope",
  "coordinates": [[ -0.087890625, 51.50874245880332 ], [
    2.4169921875, 48.80686346108517 ]]
}
```

3. 多边形

一个多边形通过一个连接点的列表来创建。数组中的第一个点和最后一个点必须一样，从而让该多边形是闭合的。一个例子如下：

```
{
  "type": "polygon",
  "coordinates": [[
     [-5.756836, 49.991408],
     [-7.250977, 55.124723],
     [1.845703, 51.500194],
     [-5.756836, 49.991408]
  ]]
}
```

仔细查看形状的定义，你会发现一个它是一个可扩展的数组。归功于此，你可以定义多个多边形。在这种情况下，第一个多边形定义基准形状，其他多边形将从基准形状中排除。

4. 多个多边形

我们可以创建一个包含多个多边形（Multipolygon）的形状，例子如下所示：

```
{
  "type": "multipolygon",
  "coordinates": [
```

6

```
      [[
        [-5.756836, 49.991408],
        [-7.250977, 55.124723],
        [1.845703, 51.500194],
        [-5.756836, 49.991408]
      ]],
      [[
        [-0.087890625, 51.50874245880332],
        [2.4169921875, 48.80686346108517],
        [3.88916015625, 51.01375465718826],
        [-0.087890625, 51.50874245880332]
      ]]
    ]
  }
```

multipolygon形状包含多个多边形，与polygon类型的规则一样。因此，我们可以有多个多边形，此外还可以包含多个被排除的形状。

5. 一个示例用法

现在，我们的索引中有了geo_shape字段，以检查哪些城市位于英国。为此可以发送下列查询：

```
{
  "filter": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "polygon",
          "coordinates": [[
            [-5.756836, 49.991408], [-7.250977, 55.124723], [-
              3.955078, 59.352096], [1.845703, 51.500194], [-
              5.756836, 49.991408]
          ]]
        }
      }
    }
  }
}
```

polygon类型以非常精确的方式定义了英国的边界。Elasticsearch返回如下的响应：

```
"hits": [
  {
    "_index": "map2",
    "_type": "poi",
    "_id": "2",
    "_score": 1,
    "_source": {
      "name": "London",
      "location": {
        "type": "point",
        "coordinates": [
          -0.1275,
```

```
            51.507222
          ]
        }
      }
    }
]
```

据我们所知，该结果是正确的。

6. 索引中形状的排序

通常，形状的定义很复杂，定义的区域也不会经常改变（例如，英国的边界）。在这种情况下，如果能在索引中定义形状并且在查询中使用它们，就会很方便。这是可能的，我们现在来讨论如何做到这一点。像往常一样，先定义一个合适的映射，如下所示：

```
{
  "country": {
    "properties": {
      "name": { "type": "string", "index": "not_analyzed" },
      "area": { "type": "geo_shape" }
    }
  }
}
```

该映射类似于以前使用的映射。我们只是改变了字段名称。使用的示例数据如下：

```
{"index": { "_index": "countries", "_type": "country", "_id": 1 }}
{"name": "UK", "area": {"type": "polygon", "coordinates": [[ [-
  5.756836, 49.991408], [-7.250977, 55.124723], [-3.955078,
  59.352096], [1.845703, 51.500194], [-5.756836, 49.991408] ]]}}
{"index": { "_index": "countries", "_type": "country", "_id": 2 }}
{"name": "France", "area": { "type":"polygon", "coordinates": [ [
  [ 3.1640625, 42.09822241118974 ], [ -1.7578125,
  43.32517767999296 ], [ -4.21875, 48.22467264956519 ], [
  2.4609375, 50.90303283111257 ], [ 7.998046875,
  48.980216985374994 ], [ 7.470703125, 44.08758502824516 ], [
  3.1640625, 42.09822241118974 ] ] ] }}
{"index": { "_index": "countries", "_type": "country", "_id": 3 }}
{"name": "Spain", "area": { "type": "polygon", "coordinates": [ [
  [ 3.33984375, 42.22851735620852 ], [ -1.845703125,
  43.32517767999296 ], [ -9.404296875, 43.19716728250127 ], [ -
  6.6796875, 41.57436130598913 ], [ -7.3828125, 36.87962060502676
  ], [ -2.109375, 36.52729481454624 ], [ 3.33984375,
  42.22851735620852 ] ] ] }}
```

你可以看到数据中，每个文档包含一个polygon类型。多边形定义给定国家的区域（不准确）。记住形状的第一个点和最后一个点必须一样，从而形状能够闭合。现在，我们修改查询以包含索引中的形状。新查询如下所示：

```
{
  "filter": {
```

```
        "geo_shape": {
          "location": {
            "indexed_shape": {
              "index": "countries",
              "type": "country",
              "path": "area",
              "id": "1"
            }
          }
        }
      }
    }
```

比较这两种查询时，我们可以注意到shape对象变成了indexed_shape。我们需要告诉Elasticsearch到哪里去找到这个形状。为此，可以定义索引（index属性，默认是shape），类型（type属性），以及路径（path属性，默认是shape）。缺少形状的id属性，在我们的例子中，它是1。然而，如果你要索引更多形状，我们会建议你除了标识符以外，还要索引形状的名字。

6.7　卷动 API

假设我们有一个含数百万文档的索引。我们已经知道如何构建查询，何时使用过滤器等。但从查询日志中，我们看到一种特定类型的查询明显比别的查询慢。这些查询可能使用了分页，从from参数看到偏移量是一个很大的值。从应用程序的角度，这意味着用户可能在检索一个数量非常大的结果集。通常这是不合理的：如果用户没有在前几页找到想要的结果，通常会放弃。因为这个特定的行为可能不是好事（也许是窃取数据），许多应用程序限制分页到只有几十页。在我们的例子中，我们假设不是这个场景，而是必须提供此功能。

6.7.1　问题定义

Elasticsearch生成响应时，它必须确定文档的顺序来形成结果。如果我们在第一页，这不是个大问题，Elasticsearch只是找到相关文档集并收集开头的比方说20个文档。但是如果我们在第十页上，Elasticsearch要取得从第一页到第十页的所有文档，然后丢弃一到九页上的文档。该问题不是Elasticsearch特有的，类似的情况会发生在每一个使用所谓的优先队列的系统中，比如，数据库系统。

6.7.2　作为解决方案的卷动

解决方案很简单。既然Elasticsearch必须为每个请求做一些操作（确定某页面之前的文档），我们可以让Elasticsearch为后续查询存储该信息。缺点是不能永远存储这些信息，因为资源有限。Elasticsearch假定我们可以声明需要此信息多久可用。来看看它实际是怎么工作的。

首先，像平常一样查询Elasticsearch。然而，除了所有已知的参数，再添加一个参数，它带有我们使用卷动的信息，以及我们建议Elasticsearch保存结果信息多长时间。为此可以发送如下查询：

```
curl 'localhost:9200/library/_search?pretty&scroll=5m' -d '{
  "query" : {
    "match_all" : { }
  }
}'
```

这个查询的具体内容无关紧要。重要的是Elasticsearch如何修改响应。看看Elasticsearch返回的响应中的前几行：

```
{
  "_scroll_id" :
    "cXVlcnlUaGVuRmV0Y2g7NTsxMDI6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMD
    U6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMDQ6dklNMlkzTG1RTDJ2b25oTDNEN
    mJzZzsxMDE6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMDM6dklNMlkzTG1RTDJ
    2b25oTDNENmJzZzswOw==",
    "took" : 9,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
  },
  "hits" : {
    "total" : 1341211,
    ...
```

新的部分是`_scroll_id`节点。这是一个我们在后续查询中要使用的句柄。Elasticsearch对此有一个特殊的端点：`_search/scroll`。我们来看看下面的例子：

```
curl -XGET
  'localhost:9200/_search/scroll?scroll=5m&pretty&scroll_id=
  cXVlcnlUaGVuRmV0Y2g7NTsxMjg6dklNlkzTG1RTDJ2b25oTDNENmJzZzsxMjk6
  dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMzA6dklNMlkzTG1RTDJ2b25oTDNENmJzZ
  zsxMjc6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMjY6dklNMlkzTG1RTDJ2b25oT
  DNENmJzZzswOw=='
```

现在，每一次使用`scroll_id`对该端点进行调用，都会返回结果的下一页。记住，这个句柄只在定义的不活动时间内有效。时间过去后，对无效`scroll_id`的查询将返回一个错误响应，类似下面这样：

```
{
  "_scroll_id" :
  "cXVlcnlUaGVuRmV0Y2g7NTsxMjg6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMj
  k6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMzA6dklNMlkzTG1RTDJ2b25oTDNEN
  mJzZzsxMjc6dklNMlkzTG1RTDJ2b25oTDNENmJzZzsxMjY6dklNMlkzTG1RTDJ2
  b25oTDNENmJzZzswOw==",
```

6

```
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 0,
    "failed" : 5,
    "failures" : [ {
      "status" : 500,
      "reason" : "SearchContextMissingException[No search context
      found for id [128]]"
    }, {
      "status" : 500,
      "reason" : "SearchContextMissingException[No search context
      found for id [126]]"
    }, {
      "status" : 500,
      "reason" : "SearchContextMissingException[No search context
      found for id [127]]"
    }, {
      "status" : 500,
      "reason" : "SearchContextMissingException[No search context
      found for id [130]]"
    }, {
      "status" : 500,
      "reason" : "SearchContextMissingException[No search context
      found for id [129]]"
    } ]
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  }
}
```

当然，这种解决方案并不理想，当有很多对各种结果的随机页面的请求时，或请求之间的时间很难确定时，它就不是很适合。但是，当你想获得更大的结果集，如多个系统之间传输数据时，可以成功地运用该方案。

6.8 多词条过滤器

Elasticsearch中，多词条过滤器乍看是个非常简单的过滤器。使用最简单的形式，可以过滤那些与给定的未经分析的词条之一匹配的文档。一个使用多词条过滤器的例子如下：

```
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "terms" : {
          "title" : [ "crime", "punishment" ]
```

```
          }
        }
      }
    }
  }
```

上述查询将找到title字段与crime或者punishment词条匹配的文档。多词条过滤器的工作方式是遍历所提供的词条，并查找跟这些词条匹配的文档。当然，匹配文档的标识符被加载到一个叫bitset的结构中并被缓存起来。有时，我们可能希望改变此默认行为。为此可以提供execution参数，设成以下所示的某个值。

- plain：这是默认方法，遍历提供的所有词条，把结果存在bitset中并缓存。
- fielddata：这个方法生成词条过滤器，使用fielddata缓存来比较词条。过滤已经加载到fielddata缓存中的字段时，此模式是非常有效的。比如，用来排序、切面或者使用索引预热器预热的字段，都会被加载到fielddata缓存中。该执行模式对在大量词条上过滤非常有效。
- bool：此方法为每个词条生成一个词条过滤器，并且把它们构建成bool过滤器。构建的bool过滤器本身不被缓存，因为它执行那些用来构建它的词条过滤器，而这些过滤器已经被缓存。
- and：此方法跟bool方法类似，只是Elasticsearch构建and过滤器而不是bool过滤器。
- or：此方法跟bool方法类似，只是Elasticsearch构建or过滤器而不是bool过滤器。

一个使用execution参数的示例查询如下所示：

```
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "terms" : {
          "title" : [ "crime", "punishment" ],
          "execution" : "and"
        }
      }
    }
  }
}
```

词条查找

我们讨论多词条过滤器不是因为它过滤文档的能力，而是因为Elasticsearch 0.90.6添加的词条查找功能。词条查找机制可以用来从提供的源中加载词条，而不是显式地传入词条列表。为了说明它是如何工作的，创建一个新的索引，并索引3个文档，使用下面的命令：

```
curl -XPOST 'localhost:9200/books/book/1' -d '{
  "id" : 1,
```

```
  "name" : "Test book 1",
  "similar" : [ 2, 3 ]
}'
curl -XPOST 'localhost:9200/books/book/2' -d '{
  "id" : 2,
  "name" : "Test book 2",
  "similar" : [ 1 ]
}'
curl -XPOST 'localhost:9200/books/book/3' -d '{
  "id" : 3,
  "name" : "Test book 3",
  "similar" : [ 1, 3 ]
}'
```

现在，假设我们想得到类似于标识符等于3那本书的所有书。当然，可以先获取第3本书，得到`similar`字段的值，然后执行另一个查询。但让我们使用词条查找功能；基本上，我们会让Elasticsearch检索文档并加载`similar`字段的值。为此，可以执行以下命令：

```
curl -XGET 'localhost:9200/books/_search?pretty' -d '{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "terms" : {
          "id" : {
            "index" : "books",
            "type" : "book",
            "id" : "3",
            "path" : "similar"
          },
          "_cache_key" : "books_3_similar"
        }
      }
    }
  },
  "fields" : [ "id", "name" ]
}'
```

上面命令的响应将如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
```

```
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0,
      "fields" : {
        "id" : 1,
        "name" : "Test book 1"
      }
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "3",
      "_score" : 1.0,
      "fields" : {
        "id" : 3,
        "name" : "Test book 3"
      }
    } ]
  }
}
```

在前面的响应中，你可以看到，我们得到了想要的书：标识符为1和3的两本。当然，词条查找机制被高度优化：如果信息存在，它将使用缓存信息。此外，_cache_key属性用于指定缓存查找结果的键值。建议设置该值，以便需要时可以轻松地清除缓存。当然_cache_key属性值在不同的查询中应该不同。

> 注意，应该存储_source字段，词条查找功能才能工作。

1. 词条查找的查询结构

回忆一下我们使用的词条查找过滤器，以便完全理解要讨论的查询：

```
"filter" : {
  "terms" : {
    "id" : {
      "index" : "books",
      "type" : "book",
      "id" : "3",
      "path" : "similar"
    },
    "_cache_key" : "books_3_similar"
  }
}
```

我们用了一个简单的过滤查询，有一个匹配所有文档的查询和一个多词条过滤器。因为过滤器中把其他属性分组在一起的对象名字是id，因此它使用id字段来过滤文档。除了id，我们使用以下属性。

- index：这指定了我们从哪个索引加载词条。在我们的例子中，是books索引。
- type：这指定了我们感兴趣的类型，在我们例子中是book类型。
- id：这指定了应该从中读取词条列表的文档的标识符。在我们的例子中，是标识符为3的文档。
- path：这指定了应该从中加载词条的字段名，在我们的查询中是similar字段。

并且还可以使用另外两个属性。

- routing：此选项指定了加载词条到过滤器时，Elasticsearch应该使用的路由值。
- cache：此选项指定了Elasticsearch是否缓存从加载文档生成的过滤器。默认情况下，它设置为true，意味着Elasticsearch将缓存过滤器。

> 注意当使用词条查找机制时，不考虑execution属性。

2. 词条查找的缓存设置

Elasticsearch允许我们设置词条查找机制使用的缓存。为了控制该缓存，可以在elasticsearch.yml文件中设置如下属性。

- indices.cache.filter.terms.size：默认为10mb。指定了Elasticsearch用在词条查找缓存上的最大内存值。大多数情况下，默认值已经足够，但如果指定加载非常大的数据，你可以增加这个值。
- indices.cache.filter.terms.expire_after_access：它指定了缓存项被最后一次访问后的最大失效时间。默认不启用。
- indices.cache.filter.terms.expire_after_write：它指定了缓存项被放到缓存后的最大失效时间。默认不启用。

6.9　小结

本章展示了更多Elasticsearch的数据分析能力。我们使用聚合和切面给索引中的数据带来意义；通过使用Elasticsearch建议器，为应用程序引入了拼写检查和自动完成功能；通过使用预匹配器，创建了警报功能，并且使用附件功能把二进制文件编入索引。我们对地理空间数据进行索引和搜索，使用卷动API有效获取大量结果。最后，使用词条查找机制，加快了提取词条列表的查询过程。

下一章将聚焦Elasticsearch集群以及如何处理它们。读者将看到什么是节点发现，它是如何使用的，以及如何改变它的配置。读者将了解时光之门和恢复模块，并改变它们的配置。也会看到Elasticsearch的缓冲区是什么，它们在哪里使用，以及如何配置它们。届时将准备一个高索引和高查询吞吐量的集群，并且将使用索引模板和动态映射。

第 7 章

深入Elasticsearch集群

上一章介绍了更多Elasticsearch的数据分析能力，使用聚合和切面为索引中的数据赋予意义，还通过使用Elasticsearch建议器为应用程序引入了拼写检查和自动完成功能。通过使用预匹配器，我们创建了警报功能，并且使用附件功能把二进制文件编入索引。我们索引和搜索地理空间数据，并使用卷动API有效获取大量的结果。最后，使用词条查找机制，加快了提取词条列表的查询过程。在本章，你将学习以下内容：

- 理解节点的发现机制、配置和调优；
- 恢复和时光之门模块的控制；
- 为高查询和高索引用例准备Elasticsearch；
- 使用索引模板和动态映射。

7.1　节点发现

启动一个Elasticsearch节点时，该节点会开始寻找具有相同集群名字并且可见的主节点。如果找到主节点，该节点加入一个已经组成了的集群；如果没有找到，该节点成为主节点（如果配置允许）。形成集群和寻找节点的过程称为发现。负责发现的模块有两个主要目的：选出一个主节点和发现集群中的新节点。本节将讨论如何配置和优化发现模块。

7.1.1　发现的类型

默认在没有安装额外插件的情况下，Elasticsearch允许使用zen发现，它提供了多播和单播发现。在计算机网络术语中，多播（http://en.wikipedia.org/wiki/Multicast）是指在单个传输中将消息传递到一组计算机中。单播（http://en.wikipedia.org/wiki/Unicast）指的是一次只通过网络传输单条消息到单个主机上。

> 当使用多播发现时，Elasticsearch试图找到所有能够接收和响应多播消息的节点。如果使用单播方法，你需要提供集群中的至少一部分主机，节点会尝试连接到它们。

选择多播还是单播，首先你应该知道你的网络能否处理多播消息。如果它可以，使用多播将更简单。如果你的网络不能处理多播，请使用单播发现。另一个使用单播发现的原因是安全：不想任何节点误加入你的集群中。所以，如果要运行多个集群，或者开发人员的计算机和集群在同一个网络中，使用单播是更好的选择。

> 如果你使用的是Linux操作系统，并希望检查你的网络是否支持多播，请对你的网络接口（通常是eth0）使用`ifconfig`命令。如果你的网络支持多播，你会从前面命令的响应中看到`MULTICAST`属性。

7.1.2　主节点

我们已经看到，发现的主要目的之一就是要选择一个主节点，它将查看整个集群。主节点会检查所有其他节点，看它们是否有响应（其他节点也ping主节点）。主节点还将接受想加入群集的新节点。如果不知什么缘故，主节点跟集群断开连接了，其余节点将从中选择一个新的主节点。所有这些过程都基于我们提供的配置自动完成。

1. 配置主节点和数据节点

默认情况下，Elasticsearch允许节点同时成为主节点和数据节点。但在特定情况下，你可能希望有只保存数据的工作节点，以及只处理请求和管理集群的主节点。一种情形是当你需要处理大量的数据，这时数据节点应该尽可能地高性能。为了把节点设置成只保存数据，需要告诉Elasticsearch，不希望这些节点成为主节点。为此，在elasticsearch.yml配置文件中添加如下属性：

```
node.master: false
node.data: true
```

为设置节点不保存数据，而只做主节点，我们希望通知Elasticsearch不希望这些节点保存数据。为此，在elasticsearch.yml配置文件中添加如下属性：

```
node.master: true
node.data: false
```

请注意，如果不设置的话，`node.master`和`node.data`默认都为`true`。但我们倾向于在配置中包含它，这样更清晰。

2. 主节点选取的配置

想象一下你创建了一个包含10个节点的集群。一切工作正常，直到有一天你的网络出现故障，有三个节点从集群中断开连接，但它们仍然能互相看见对方。由于zen发现机制和主节点选取的过程，断开的三个节点中选出了一个新的主节点，你最终有了两个名字相同的集群，各自都有一个主节点。这样的情况称为脑裂（split-brain），你必须尽可能避免。当脑裂发生时，你有两个（或更多）不会互相连接的集群，直到网络（或其他任何）问题得到修复。

为了防止脑裂发生，Elasticsearch提供了discovery.zen.minimum_master_nodes属性。该属性定义的是为了形成一个集群，有主节点资格并互相连接的节点的最小数目。现在回到我们的集群，如果把discovery.zen.minimum_master_nodes属性设置为所有可用节点个数加1的50%（在我们的例子中即是6），将只会有一个集群。为什么呢？因为如果网络正常，我们将有10个节点，多于6个，这些节点将成为一个集群。3个节点断开连接后，第一个集群仍然在运行。然而，因为只有3个断开连接，小于6个，它们将无法选出一个新的主节点，只能等待重新连回原来的集群。

7.1.3 设置集群名

如果我们不在elasticsearch.yml文件设置cluster.name属性，Elasticsearch将使用默认值：elasticsearch。这不见得很好，因此建议你设置cluster.name属性为你想要的名字。如果你想在一个网络中运行多个集群，也必须设置cluster.name属性，否则，将导致不同集群的所有节点都连接在一起。

1. 配置多播

多播是zen发现的默认方法。除了常见的设置（很快就会讨论到）外，我们还可以控制以下四个属性。

- discovery.zen.ping.multicast.group：用于多播请求的群组地址，默认为224.2.2.4。
- discovery.zen.ping.multicast.port：用于多播通信的端口号，默认为54328。
- discovery.zen.ping.multicast.ttl：多播请求被认为有效的时间，默认为3秒。
- discovery.zen.ping.multicast.address：Elasticsearch应该绑定的地址。默认为null，意味着Elasticsearch将尝试绑定到操作系统可见的所有网络接口。

要禁用多播，应在elasticsearch.yml文件中添加discovery.zen.ping.multicast.enabled属性，并且把值设为false。

2. 配置单播

因为单播的工作方式，我们需要指定至少一个接收单播消息的主机。为此，在elasticsearch.yml文件中添加discovery.zen.ping.unicast.hosts属性。基本上，我们应该在discovery.zen.ping.unicast.hosts属性中指定所有形成集群的主机。指定所有主机不是必须的，只需要提供足够多的主机以保证最少有一个能工作。如果希望指定192.168.2.1、192.168.2.2和192.168.2.3主机，可以用下面的方法来设置上述属性：

```
discovery.zen.ping.unicast.hosts: 192.168.2.1:9300, 192.168.2.2:9300,
192.168.2.3:9300
```

7

你也可以定义一个Elasticsearch可以使用的端口范围，例如，从9300到9399端口，指定以下命令行：

```
discovery.zen.ping.unicast.hosts: 192.168.2.1:[9300-9399],
192.168.2.2:[9300-9399], 192.168.2.3:[9300-9399]
```

请注意，主机之间用逗号隔开，并指定了预计用于单播消息的端口。

> 使用单播时，总是设置`discovery.zen.ping.multicast.enabled`为`false`。

7.1.4　节点的ping设置

除了刚刚讨论的设置，还可以控制或改变默认的ping设置。ping是一个节点间发送的信号，用来检测它们是否还在运行以及可以响应。主节点会ping集群中的其他节点，其他节点也会ping主节点。可以设置下面的属性。

- `discovery.zen.fd.ping_interval`：该属性默认为1s（1秒钟），指定了节点互相ping的时间间隔。
- `discovery.zen.fd.ping_timeout`：该属性默认为30s（30秒钟），指定了节点发送ping信息后等待响应的时间，超过此时间则认为对方节点无响应。
- `discovery.zen.fd.ping_retries`：该属性默认为3，指定了重试次数，超过此次数则认为对方节点已停止工作。

如果你遇到网络问题，或者知道你的节点需要更多的时间来等待ping响应，可以把上面那些参数调整为对你的部署有利的值。

7.2　时光之门与恢复模块

除了我们的索引和索引里面的数据，Elasticsearch还需要保存类型映射和索引级别的设置等元数据。此信息需要被持久化到别的地方，这样就可以在群集恢复时读取。这就是为什么Elasticsearch引入了时光之门模块。你可以把它当做一个集群数据和元数据的安全的避风港。你每次启动群集，所有所需数据都从时光之门读取，当你更改你的集群，它使用时光之门模块保存。

7.2.1　时光之门

为了设置我们希望使用的时光之门类型，需要在elasticsearch.yml配置文件中添加`gateway.type`属性，并设置为`local`。目前，Elasticsearch推荐使用本地时光之门类型（`gateway.type`设为`local`），这也是默认值。过去有其他时光之门类型（比如`fs`、`hdfs`和`s3`），但已被弃用，

将在未来的版本中删除。因此跳过对它们的讨论。默认的本地时光之门类型在本地文件系统中存储索引和它们的元数据。跟其他时光之门类型相比，这种的写操作不是异步的。所以，每当写操作成功，都可以确保数据已经被写入了gateway（也就是说，它被索引或存储在事务日志中）。

7.2.2 恢复控制

除了选择时光之门类型以外，Elasticsearch允许配置何时启动最初的恢复过程。恢复是初始化所有分片和副本的过程，从事务日志中读取所有数据，并应用到分片上。基本上，这是启动Elasticsearch所需的一个过程。

假设有一个由10个Elasticsearch节点组成的集群。应该通知Elasticsearch我们的节点数目，设置gateway.expected_nodes属性为10。我们告知Elasticsearch有资格来保存数据且可以被选为主节点的期望节点数目。集群中节点的数目等于gateway.expected_nodes属性值时，Elasticsearch将立即开始恢复过程。

也可以在8个节点之后开始恢复，设置gateway.recover_after_nodes属性为8。可以将它设置为任何我们想要的值，但应该把它设置为一个值以确保集群状态快照的最新版本可用，一般在大多数节点可用时开始恢复。

然而，还有一件事：我们希望gateway在集群形成后的10分钟以后开始恢复，因此设置gateway.recover_after_time属性为10m。这个属性告诉gateway模块，在gateway.recover_after_nodes属性指定数目的节点形成集群之后，需要等待多长时间再开始恢复。如果我们知道网络很慢，想让节点之间的通信变得稳定时，可能需要这样做。

上述属性值都应该设置在elasticsearch.yml配置文件中，如果想设置上面的值，则最终在文件中得到下面的片段：

```
gateway.recover_after_nodes: 8
gateway.recover_after_time: 10m
gateway.expected_nodes: 10
```

额外的gateway恢复选项

除了提到的选项，Elasticsearch还允许做一些控制。额外的选项如下。

- gateway.recover_after_master_nodes：这个属性跟gateway_recover_after_nodes属性类似。它指定了多少个有资格成为主节点的节点在集群中出现时才开始启动恢复，而不是指定所有节点。
- gateway.recover_after_data_nodes：这个属性也跟gateway_recover_after_nodes属性类似。它指定了多少个数据节点在集群中出现时才开始启动恢复。

- gateway.expected_master_nodes：这个属性跟gateway.expected_nodes属性类似，它指定了希望多少个有资格成为主节点的节点出现，而不是所有的节点。
- gateway.expected_data_nodes：这个属性也跟gateway.expected_nodes属性类似，它指定了你期望出现在集群中的数据节点的个数。

7.3　为高查询和高索引吞吐量准备 Elasticsearch 集群

直到现在，我们谈的主要是Elasticsearch在处理查询和索引数据方面不同的功能。在这里，我们想简要谈谈为高查询和高索引吞吐量准备Elasticsearch集群。本节的开头先提到一些还没谈到的Elasticsearch功能，它们对优化集群很重要。我们知道，这是一个非常简要的介绍，但我们会试着只介绍那些我们认为重要的内容。之后，会就如何调整这些功能给出一般性建议，以及注意事项。希望通过阅读这一节，你能够在调优集群时知道要注意的事项。

7.3.1　过滤器缓存

过滤器缓存负责缓存查询中使用到的过滤器。你可以从缓存中飞快地获取信息。如果设置得当，它将有效地提高查询速度，尤其是那些包含已经执行过的过滤器的查询。

Elasticsearch包含两种类型的过滤器缓存：节点过滤器缓存（默认）和索引过滤器缓存。节点过滤器缓存被分配在节点上的所有索引共享，可以配置成使用特定大小的内存，或分配给Elasticsearch总内存的百分比。为了设置这个值，应该包含indices.cache.filter.size这个节点属性值，并设置成需要的大小或百分比。

第二种过滤器缓存基于索引级别。一般来说，你应该使用节点级别的过滤器缓存，因为很难预测每个索引的最终缓存大小，通常也不知道最终节点上会有多少索引。我们将省略对索引级别过滤器缓存的进一步解释，关于它的更多信息可以在官方文档找到，或者我们的书*Mastering ElasticSearch*。

7.3.2　字段数据缓存和断路器

字段数据缓存是Elasticsearch缓存的一部分，主要用于当查询对字段执行排序或切面时。Elasticsearch把用于该字段的数据加载到内存，以便基于每个文档快速访问这些值。构建这些字段数据缓存是昂贵的，所以最好有足够的内存，以便缓存中的数据一旦加载就留在缓存中。

> 你也可以把字段配置成使用doc值，而不是字段数据缓存。doc值在2.2节中讨论过。

允许用于字段数据缓存的内存大小可以用indices.fielddata.cache.size属性来控制。可以把它设置为绝对值（例如2 GB）或者分配给Elasticsearch实例的内存百分比（例如40%）。请注意，这些值是节点级别，而不是索引级别的。为其他条目丢弃部分缓存会导致查询性能变差，所以建议要有足够的物理内存。此外请记住，默认情况下，字段数据缓存的大小是无限的，所以如果我们不小心，会导致集群的内存爆炸。

我们还可以控制字段数据缓存的过期时间，同样，默认情况下字段数据缓存永远不过期。可以使用indices.fielddata.cache.expire属性来控制，将其设置为最大的不活动时间。例如，将它设置为10m将导致缓存不活动10分钟后过期。记住重建字段数据缓存是非常昂贵的，一般情况下，你不应该设置过期时间。

断路器

字段数据断路器（field data circuit breaker）允许估计一个字段加载到缓存所需的内存。利用它，可以通过抛出异常来防止一些字段加载到内存中。Elasticsearch有两个属性来控制断路器的行为。第一个是indices.fielddata.breaker.limit属性，默认值为80%，可以使用集群的更新设置API来动态地修改它。这意味着，当查询导致加载字段的值所需的内存超过了Elasticsearch进程中可用堆内存的80%时，将引发一个异常。第二个属性是indices.fielddata.breaker.overhead，默认为1.03，它定义了用来与原始估计相乘的一个常量。

7.3.3 存储模块

Elasticsearch中的存储模块负责控制如何写入索引数据。我们的索引可以完全存储在内存或者一个持久化磁盘中。纯内存的索引极快但不稳定，而基于磁盘的索引慢一些，但可容忍故障。

利用index.store.type属性，可以指定使用哪种存储类型，可用的选项包括下面这些。

- simplefs：这是基于磁盘的存储，使用随机文件来访问索引文件。它对并发访问的性能不够好，因此不建议在生产环境使用。
- niofs：这是第二个基于磁盘的索引存储，使用Java NIO类来访问索引文件。它在高并发环境提供了非常好的性能，但不建议在Windows平台使用，因为Java在这个平台的实现有bug。
- mmapfs：这是另一个基于磁盘的存储，它在内存中映射索引文件（对于mmap，请参阅http://en.wikipedia.org/wiki/Mmap）。这是64位系统下的默认存储，因为为索引文件提供了操作系统级别的缓存，因此它的读操作更快。你需要确保有足够数量的虚拟地址空间，但在64位系统下，这不是问题。
- memory：这将把索引存在内存中。请记住你需要足够的物理内存来存储所有的文档，否则Elasticsearch将失败。

7.3.4　索引缓冲和刷新率

当说到索引时，Elasticsearch允许你为索引设置最大的内存数。indices.memory.index_buffer_size属性可以控制在一个节点上，所有索引的分片共拥有的最大内存大小（或者最大堆内存的百分比）。例如，把该属性设置为20%，将告诉Elasticsearch提供最大堆大小20%的内存给索引缓冲。

此外，还有个indices.memory.min_shard_index_buffer_size属性，默认为4mb，允许为每个分片设置最小索引缓冲。

索引刷新率

有关索引的最后一件事是index.refresh_interval属性，它指定在索引上，默认为1s（1秒钟），指定了索引搜索器对象刷新的频率，基本上意味着数据视图刷新的频率。刷新率越低，文档对搜索操作可视的时间越短，也意味着Elasticsearch将需要利用更多资源来刷新索引视图，因此索引和搜索操作将会变慢。

> 对庞大的批量索引，例如，当对数据重建索引时，建议在索引阶段把index.refresh_interval属性设为-1。

7.3.5　线程池的配置

Elasticsearch使用多个池来控制线程的处理，以及控制用户请求占用多少内存消耗。

> Java虚拟机允许应用程序使用多线程并行地运行程序的多个分支。有关Java线程的更多信息，请参阅http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html。

我们尤其感兴趣的是Elasticsearch公开的如下线程池类型。

- cache：这是无限制的线程池，为每个传入的请求创建一个线程。
- fixed：这是一个有着固定大小的线程池，大小由size属性指定，允许你指定一个队列（使用queue_size属性指定）用来保存请求，直到有一个空闲的线程来执行请求。如果Elasticsearch无法把请求放到队列中（队列满了），该请求将被拒绝。

有很多线程池（可以使用type属性指定要配置的线程类型），然而，对于性能来说，最重要的是下面几个。

- index：此线程池用于索引和删除操作。它的类型默认为fixed，size默认为可用处理器的数量，队列的size默认为300。

- search：此线程池用于搜索和计数请求。它的类型默认为fixed，size默认为可用处理器的数量乘以3，队列的size默认为1000。
- suggest：此线程池用于建议器请求。它的类型默认为fixed，size默认为可用处理器的数量，队列的size默认为1000。
- get：此线程池用于实时的GET请求。它的类型默认为fixed，size默认为可用处理器的数量，队列的size默认为1000。
- bulk：你可以猜到，此线程池用于批量操作。它的类型默认为fixed，size默认为可用处理器的数量，队列的size默认为50。
- percolate：此线程池用于预匹配器操作。它的类型默认为fixed，size默认为可用处理器的数量，队列的size默认为1000。

举个例子，把用于索引操作的线程池配置成fixed类型，大小为100，队列大小为500，我们将在elasticsearch.yml配置文件中设置如下属性：

```
threadpool.index.type: fixed
threadpool.index.size: 100
threadpool.index.queue_size: 500
```

记住，线程池的配置可以使用集群的更新API来更新，如下所示：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "threadpool.index.type" : "fixed",
    "threadpool.index.size" : 100,
    "threadpool.index.queue_size" : 500
  }
}'
```

7.3.6　结合起来，一些通用建议

现在，我们知道了Elasticsearch所公开的缓存和缓冲区，可以尝试结合这些知识来配置一个高索引和查询吞吐量的集群。接下来的两个小节将讨论在设置集群时，什么可以在默认配置中更改，什么是要注意的。

在讨论Elasticsearch特定配置相关的所有事情之前，应该记住，必须给予Elasticsearch足够的内存，而且是物理内存。一般来说，运行Elasticsearch的JVM进程不应该超过可用内存的50%或60%，这样做是因为要留一些可用内存给操作系统以及操作系统的I/O缓存。

然而，需要记住，50%到60%不一定总是对的。你可以想象一个有256 GB内存的节点，在节点上有个总共30 GB的索引，在这种情况下，即使分配多于60%的物理内存给Elasticsearch，也会给操作系统留下足够的内存。另外，把Xmx和Xms参数设置为相同的值以避免JVM堆的大小调整，也是个好主意。

当优化你的系统时，记得有一个在相同环境可以反复跑的性能测试。一旦你做出更改，你需要看到它是如何影响整体性能的。此外，Elasticsearch可扩展，正因为此，有时最好在单机上

做一个简单的性能测试，看看性能如何，可以从中得到什么。这样的一些观察是进一步调优的好起点。

在继续之前，注意我们不能给你高索引高查询吞吐量的秘方，因为每个部署都是不同的。因此，我们将只讨论你在调优时应该注意什么。如果你对这样的用例感兴趣，可以访问博客http://blog.sematext.com，有些作者会写一些关于性能测试的文章。

1. 选择正确的存储

当然，除了已经谈过的物理内存外，应该选择正确的存储实现。一般来说，如果运行的是64位操作系统，你应该选择`mmapfs`。如果没有运行64位操作系统，为UNIX系统选择`niofs`，为Windows系统选择`simplefs`。如果你可以容忍一个易失的存储，但希望它非常快，可以看看`memory`存储，它会给你最好的索引访问性能，但需要足够的内存来处理所有索引文件、索引和查询。

2. 索引刷新率

应该注意的第二件事是索引刷新率。我们知道刷新率指定文档多快可以对搜索操作可见。等式非常简单：刷新率越快，查询越慢，索引吞吐量越低。如果我们允许有一个较慢的刷新率，如10s或30s，设置它是不错的。这使得Elasticsearch承受的压力更少，因为内部对象重新打开的频率更低，因此，将有更多的资源用于索引和查询。

3. 优化线程池

强烈建议调整默认线程池，尤其是查询操作。在性能测试之后，你通常看到集群上的查询不堪重负，这时应该开始拒绝请求。我们认为在大多数情况下，最好是立刻拒绝该请求，而不是把它放到队列并强制应用程序等待很长时间请求处理。我们真的很想给你一个准确的数字，但这仍然在很大程度上取决于你的部署，给不了通用的建议。

4. 优化合并过程

合并过程同样在很大程度上取决于你的用例，以及若干因素，例如是否正在索引、你添加了多少数据以及做这些操作的频率。一般来说，记住查询多个段跟查询数量更少的段相比更慢。但是，想要段的数目更少，你需要付出更多的合并代价。

2.5节讨论过段合并。我们还提到了调节，它允许限制I/O操作。

通常来说，如果你想查询更快，应该以索引中更少的段为目标。如果想索引更快，应该有更多的段。如果你想兼具两者，就要找到两者之间的黄金点，让合并不会太频繁但又不会导致大量的段。使用并行合并调度器并调整默认调节值，使I/O子系统不会被合并淹没。

5. 字段数据缓存和断路器

默认情况下，Elasticsearch中的字段数据缓存是无限的。这很危险，尤其在很多字段上使用切面或排序时。如果字段基数很高，你可能会遇到更多的麻烦，麻烦的意思是说你可能会内存不足。

我们有两个不同因子可以调节，来确保不会遇到内存不足错误。首先，可以限制字段数据缓存的大小。其次是断路器，通过它可以很容易地配置成在加载过多数据时抛出一个异常。两者结合可以确保我们不会遇到内存问题。

然而，也应该记住，当数据字段缓存的大小不足以处理切面或排序请求时，Elasticsearch将从中移除数据。这将影响查询性能，因为加载字段数据信息是低效的。不过，我们认为宁愿让查询慢，也不能由于内存不足错误而导致集群不工作。

6. 索引的内存缓冲区

请记住，用于索引缓冲区的可用内存越多（`indices.memory.index_buffer_size`属性），Elasticsearch可以在内存中保存的文档也越多。但是，我们当然不想Elasticsearch占用100%的可用内存。默认情况下，该属性被设置为10%，但如果真的需要更高的索引比例，你可以提高这个百分比。我们见过一些关注数据索引的集群，把该属性设为30%，确实是有帮助的。

7. 优化事务日志

我们还没讨论到，但Elasticsearch有个内部模块称为translog（http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-translog.html）。它是分片上的结构，为预写日志（http://en.wikipedia.org/wiki/Write-ahead_logging）服务。

基本上，它允许Elasticsearch为GET操作公开最新的更新，确保数据持久性，并优化对Lucene索引的写入。

默认情况下，Elasticsearch在事务日志中保存最多5000次操作，同时最大不超过200 mb。但如果想要更高的索引吞吐量，又可以承担数据对搜索操作不可见的时间更长，就可以提高这个默认值。通过`index.translog.flush_threshold_ops`和`index.translog.flush_threshold_size`属性（两者都是索引上的设置，可以用Elasticsearch API实时更新），可以设置保存在事务日志中的最大操作数和最大的大小。我们见过一些部署把这个属性值设成默认值的10倍。

要记住一件事，如果发生故障，对于有更大事务日志的分片，其初始化当然也更慢，因为Elasticsearch需要在分片可用之前处理事务日志中的所有信息。

8. 牢记于心

当然，前面提到的因素并非全部。你应该监视Elasticsearch集群并作出相应的反应。如果你

看到索引中的段开始增长，而你不想这样，请调整合并政策。当你看到合并使用过多的I/O资源，并影响了整体性能，请调整你的调节。只是要记住，调整不是一次性的；数据会增加，查询数目也会增加，你要不断适配它。

7.4　模板和动态模板

在2.2节中，我们学过映射、如何创建它们以及类型确定机制是如何工作的。现在将进入更高级的主题，如何为新的索引动态地创建映射以及如何在模板中应用一些逻辑。

7.4.1　模板

我们在前面看到过，索引配置，特别是映射，可以是很复杂的野兽。如果有可能定义一个或多个映射，用在每个新创建的索引上，而不需要每次创建索引时都发送它们，那就太好了。Elasticsearch创作者预见到这点，并实现了一个叫索引模板（index templates）的功能。每个模板定义了一个模式，用来比较新创建索引的名称。当两者匹配，在模板中定义的值复制到索引的结构定义中。当多个模板匹配新创建索引的名称时，所有模板都会被应用，后应用的模板中的值将覆盖先应用的模板中定义的值。这非常方便，因为可以在通用模板中定义一些常用设置，然后在专有模板中修改它们。此外，还有个order参数，可以强制所需模板的顺序。你可以把模板想象成一个动态映射，但它不是应用到文档中的类型，而是应用到索引。

1. 模板的一个例子

来看一个真实的模板例子。假设要创建许多索引，并且不希望在索引中存储源文档，以便让索引更小，而且也不需要任何副本。可以创建一个模板来满足这个需求，通过使用Elasticsearch的REST API，发送如下命令：

```
curl -XPUT http://localhost:9200/_template/main_template?pretty -d '{
  "template" : "*",
  "order" : 1,
  "settings" : {
    "index.number_of_replicas" : 0
  },
  "mappings" : {
    "_default_" : {
      "_source" : {
        "enabled" : false
      }
    }
  }
}'
```

从现在开始，所有创建的索引都将没有副本，也没有存储源文档。这是因为template参数值设成了*，意味着匹配所有索引名。注意例子中的_default_类型名字，这是一个特殊的类型

名，表明当前规则应适用于所有的文档类型。第二个有趣的事情是order参数。使用如下命令定义第二个模板：

```
curl -XPUT http://localhost:9200/_template/ha_template?pretty -d '{
  "template" : "ha_*",
  "order" : 10,
  "settings" : {
    "index.number_of_replicas" : 5
  }
}'
```

执行上述命令后，除了名字以ha_开头，所有其他新索引将有相同的行为。两个模板都被应用在这些索引中。首先应用order值更小的模板，然后下一个模板覆盖副本的设置。所以，名字以ha_开头的索引将有5个副本，并禁用源文档的存储。

2. 在文件中存储模板

模板也可以存储在文件中。默认情况下，文件应该放在config/templates目录中。例如，ha_template模板应该放在config/templates/ha_template.json文件中，内容如下所示：

```
{
  "ha_template" : {
    "template" : "ha_*",
    "order" : 10,
    "settings" : {
      "index.number_of_replicas" : 5
    }
  }
}
```

注意这个JSON的结构有点不同，它使用模板名字作为主对象的键。另外很重要的是，模板必须放在Elasticsearch的每个实例中。此外，文件中定义的模板不可用在REST API调用中。

7.4.2 动态模板

有时，我们想依赖一个字段名称和类型来定义一个类型。这是动态模板发挥作用的地方。动态模板跟通常的映射相似，但每个模板都定义了它的模式，并将其应用于文档的字段名称。如果一个字段名称与模式匹配，则使用该模板。看看下面的示例：

```
{
  "mappings" : {
    "article" : {
      "dynamic_templates" : [
      {
        "template_test": {
          "match" : "*",
          "mapping" : {
            "index" : "analyzed",
```

```
            "fields" : {
              "str": {"type": "{dynamic_type}",
                      "index": "not_analyzed" }
            }
          }
        }
      }]
    }
  }
}
```

在前面的例子中，我们为article类型定义了一个映射。在这个映射中，我们只有一个动态模板，名为template_test。由于match属性设置成了星号，这个模板将应用到输入文档的每个字段上。每个字段都将被视为多字段，由原始字段名字（比如，title）和第二个以str作为后缀的字段（比如，title.str）。第一个字段的类型由Elasticsearch决定（{dynamic_type}类型），第二个字段将是一个字符串（因为string类型）。

1. 匹配模式

我们有以下两种方式定义匹配模式。

- match：如果字段名与模式相匹配的话，则使用该模板（这是我们例子中使用的类型）。
- unmatch：如果字段名与模式不匹配，则使用该模板。

默认情况下，该模式非常简单，并且使用glob模式，可以使用match_pattern=regexp来修改。添加此属性后，可以使用正则表达式提供的所有魔法来匹配或不匹配模式。

有些变种，比如path_match和path_unmatch，可以用来匹配嵌套文档的名字。

2. 字段定义

当写入一个目标字段定义时，可以使用下面的变量。

- {name}：输入文档中找到的原始字段的名字。
- {dynamic_type}：原始文档确定的类型。

> 请注意，Elasticsearch根据定义的顺序检查模板，并应用第一个匹配的模板。这意味着最通用的模板（比如，使用"match":"*"的那些）应该定义在最后面。

7.5　小结

本章介绍了Elasticsearch的一些功能，如节点发现、模块负责什么，以及如何调优。也学习了恢复和时光之门模块如何设置以配合集群，它们提供了什么配置选项。我们还探讨了一些

Elasticsearch的内部功能，使用这些来为高索引和高查询调优集群。最后，我们使用了模板和动态映射，以便更好地管理动态索引。

下一章关注Elasticsearch的管理能力。我们将学习如何备份集群数据，使用可用的API调用监视我们的集群；讨论使用Elasticsearch API如何控制分片的分配，如何在集群中移动分片；了解什么是索引预热器，以及它们的用处；使用别名。最后，还将学习如何安装和管理Elasticsearch插件，以及用更新设置API能做些什么。

第 8 章

集群管理

上一章介绍了Elasticsearch中节点发现如何工作，如何调优，恢复和时光之门模块，如何通过调整某些Elasticsearch内部构件来准备高索引和高查询的集群及其内部构件。最后，使用索引模板和动态映射来轻松地控制动态索引的结构。在本章，你将了解以下内容：

- 使用Elasticsearch的快照功能；
- 使用Elasticsearch API监控集群；
- 调整集群的再平衡，以配合我们的需要；
- 使用Elasticsearch API移动分片；
- 预热；
- 使用别名来减轻每天的工作；
- 安装Elasticsearch插件；
- 使用Elasticsearch的更新设置API。

8.1 Elasticsearch 时光机

一个好的软件可以管理硬件故障或人为错误等异常情况。尽管几台服务器的集群很少暴露硬件问题，但不好的事情仍会发生。假设你需要恢复索引。一个可能的解决方案是取得以SQL数据库存储的所有主数据，并重新生成索引。但如果时间过长，或者更糟的是，数据只存储在Elasticsearch中，怎么办？Elasticsearch 1.0之前，创建索引的备份很不容易，过程包括关闭集群、复制数据文件等。幸好，现在可以使用快照。来看看它是如何工作的。

8.1.1 创建快照存储库

快照保存它创建的时间点上所有跟集群相关的数据，包括集群状态和索引的信息。至少在创建第一个快照之前，必须创建一个快照存储库。

每个存储库由名称区分，应该定义如下几个方面。

- `name`：这是存储库的唯一名称，后面会用到它。

- type：这是存储库的类型，可能的值包括fs（共享文件系统中的存储库）、url（一个通过URL访问的只读存储库）。
- settings：这是不同存储库类型需要的额外信息。

现在，创建一个文件系统存储库。请注意，集群中的每个节点都应该能访问这个目录。为创建一个新的文件系统存储库，可以执行下列命令：

```
curl -XPUT localhost:9200/_snapshot/backup -d '{
  "type": "fs",
  "settings": {
    "location": "/tmp/es_backup_folder/cluster1"
  }
}'
```

前面的命令创建一个名为backup的库，将备份文件存储在由location属性指定的目录中。Elasticsearch返回下列信息：

```
{"acknowledged":true}
```

与此同时，本地文件系统中新建了backup_folder目录，但还没有任何内容。

> 我们说过，第二种存储库类型是url。它需要一个url参数，而不是location，该参数指向存储库所在的地址，例如，HTTP地址。你还可以把快照存储在Amazon S3或HDFS中，使用额外的可用插件（参见https://github.com/elasticsearch/elasticsearch-cloud-aws#s3-repository 和 https://github.com/elasticsearch/elasticsearch-hadoop/tree/master/repository-hdfs）。

现在，我们有了第一个存储库，可以使用以下命令看到它的定义：

```
curl -XGET localhost:9200/_snapshot/backup?pretty
```

还可以运行以下命令检查所有存储库：

```
curl -XGET localhost:9200/_snapshot/_all?pretty
```

如果你想删除一个快照存储库，标准的DELETE命令可以帮忙：

```
curl -XDELETE localhost:9200/_snapshot/backup?pretty
```

8.1.2 创建快照

默认情况下，创建快照时，Elasticsearch取得所有的索引和集群设置（除了瞬时的那些）。你可以创建任意数量的快照，每个快照将持有从创建的时间点开始的所有信息。快照的创建用了一种巧妙的方式：仅复制新的信息。这意味着Elasticsearch知道哪些段已经存储在存储库中，不会

再保存它们。

为了创建新的快照，需要选择一个唯一的名称，并使用下面的命令：

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp1'
```

上面的命令定义了一个名为bckp1的新快照（给定名称只能有一个快照，Elasticsearch会检查它的唯一性），数据将保存在之前定义的backup库中。该命令立即返回一个响应，如下所示：

```
{"accepted":true}
```

上述响应意味着，快照的进程已经在后台开始并继续。如果你想在实际的快照被创建后才得到响应，可以添加wait_for_completion参数，如下面的例子所示：

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp2?wait_for_completion=
  true&pretty'
```

上述命令的响应展现了新创建快照的状态：

```
{
  "snapshot" : {
    "snapshot" : "bckp2",
    "indices" : [ "art" ],
    "state" : "SUCCESS",
    "start_time" : "2014-02-22T13:04:40.770Z",
    "start_time_in_millis" : 1393074280770,
    "end_time" : "2014-02-22T13:04:40.781Z",
    "end_time_in_millis" : 1393074280781,
    "duration_in_millis" : 11,
    "failures" : [ ],
    "shards" : {
      "total" : 5,
      "failed" : 0,
      "successful" : 5
    }
  }
}
```

可以看到，Elasticsearch给出了快照过程消耗的时间、它的状态和影响到的索引等信息。

额外参数

快照命令还可以接受以下额外参数。

- indices：想拍下快照的索引名称。
- ignore_unavailable：默认为true。设置为false时，意味着如果indices参数指向了不存在的索引，命令将失败。
- include_global_state：默认值为true，指集群的状态也被写入快照中（除了那些瞬时的设置）。

- partial：快照的成功与否取决于所有分片的可用性。任何一个分片不可用，快照都将失败。设置partial为true时，Elasticsearch值保存可用分片的信息，省略丢失的分片。

使用额外参数的一个示例如下：

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp?wait_for_
completion=true&pretty' -d '{ "indices": "b*", "include_global_state":
"false" }'
```

8.1.3　还原快照

我们已经完成了快照，接下来学习如何从给定的快照中恢复数据。前面说过，可以通过名称指向一个快照。可以使用下面的命令列出所有快照：

```
curl -XGET 'localhost:9200/_snapshot/backup/_all?pretty'
```

先前创建的存储库名称为backup。为了从库中还原一个名为bckp1的快照，执行下面的命令：

```
curl -XPOST 'localhost:9200/_snapshot/backup/bckp1/_restore'
```

执行这个命令的过程中，Elasticsearch取得定义在此快照中的索引，并用快照中的数据创建它们。如果索引已经存在而且未关闭，该命令将失败。这种情况下，你可能会发现只还原某些索引是很方便的，例如：

```
curl -XPOST 'localhost:9200/_snapshot/backup/bck1/_restore?pretty' -d '{
"indices": "c*"}'
```

上述命令只还原以字母c开头的索引，还有以下可用参数。

- ignore_unavailable：与创建快照时相同。
- include_global_state：与创建快照时相同。
- rename_pattern：允许你改变存储在快照中的索引的名称。归功于此，还原的索引将有一个不同的名称。此参数的值是一个正则表达式，定义了源索引的名字，如果模式与索引名字匹配，将发生名称替换。在该模式中，应当使用以rename_replacement参数中所用括号分隔的分组。
- rename_replacement：该参数和rename_pattern一起定义了目标索引名称。使用美元符号和数字可以指向rename_pattern中合适的组。

例如，由于rename_pattern=products_(.*)，只有名称以products_开头的索引将被还原。索引名字的其他部分将使用在替换部分。配合rename_replacement=items_$1，将让products_cars索引被还原成名为items_cars的索引。

8.1.4　清理：删除旧的快照

Elasticsearch把快照库的管理交给你。目前，没有自动的清理过程。但不用担心，这很简单。例如，删除之前创建的快照：

```
curl -XDELETE 'localhost:9200/_snapshot/backup/bckp1?pretty'
```

就这么简单。该命令将从backup库中删除名为bckp1的快照。

8.2　监控集群的状态和健康度

在应用程序的正常周期中，一个很重要的方面就是监控。这使系统管理员能够检测并预防可能的问题，或至少知道失败时会发生什么。

Elasticsearch提供了非常详细的信息，使你能够检查和监控单个节点或作为一个整体的集群。这包括统计数字、有关服务器的信息、节点、索引和分片。当然，也能够获得整个集群状态的有关信息。在深入提到的API细节之前，请记住，API是复杂的，我们只是描述基本的东西。我们会试着展现什么时候开始，让你在需要非常详细的信息时能知道要找什么。

8.2.1　集群健康度API

一个最基本的API是集群健康度API，它允许使用单个HTTP命令得到整个集群的状态信息。例如，执行以下命令：

```
curl 'localhost:9200/_cluster/health?pretty'
```

上述命令返回的响应示例如下所示：

```
{
  "cluster_name" : "es-book",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 4,
  "active_shards" : 4,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```

最重要的一个信息是关于集群的状态。在例子中，我们看到集群是green状态。这意味着所有分片已妥善分配，没有错误。

暂停一下，先谈谈作为一个整体的集群什么时候会完全运作。当Elasticsearch能够根据配置

分配所有分片和副本时，集群才全面运作。在这种情况下，集群在green状态。yellow状态意味着我们已经准备好处理请求，因为主分片已经分配，但部分或所有副本还没有。最后一个状态是red，这意味着至少一个主分片没有分配，因此集群还没有准备好。这意味着查询可能返回错误或不完整的结果。

前面的命令也可以用来检查某索引的健康状况。如果想要检查library和map索引的健康度，运行以下命令：

```
curl 'localhost:9200/_cluster/health/library,map/?pretty'
```

1. 控制信息细节

Elasticsearch允许指定一个特殊的level参数，把它指定为cluster（默认）、indices或shards。这样就能够控制由健康度API返回信息的细节。我们已经看过默认的行为。当设置level为indices时，除集群信息外，还将获得每个索引的健康度。参数设置为shards告诉Elasticsearch除了返回我们在示例中看到内容，还要返回每个分片的信息。

2. 额外的参数

除了level参数，还有一些额外参数用来控制健康度API的行为。

第一个参数是timeout。它允许控制命令执行的最长时间。默认值为30s，意味着健康度命令将最长等待30秒钟，然后返回。

wait_for_status参数告诉Elasticsearch返回响应时，集群应该处于什么状态。可以把它设置为green、yellow和red。例如，设置为green时，健康度API调用将返回绿色状态，或者达到timeout时间。

wait_for_nodes参数允许设置返回响应时需要多少节点可用（或者达到timeout时间）。可以设置该参数为整数值，比如3，或者一个简单等式，比如>=3（大于或等于3个节点），<=3（小于或等于3个节点）。

最后一个参数是wait_for_relocating_shard，默认不指定。它告诉Elasticsearch应该重定位多少分片（或者等待timeout时间）。设置该参数为0意味着Elasticsearch应该等待所有重定位分片。

使用以上参数的健康度命令的一个例子如下所示：

```
curl 'localhost:9200/_cluster/health?wait_for_status=green&wait_for_
nodes=>=3&timeout=100s'
```

8.2.2 索引统计API

Elasticsearch索引是保存数据的地方，它对大多数部署来说是非常重要的部分。使用_stats

端点上的索引统计API，可以得到关于集群中索引的各种信息。当然，和Elasticsearch大多数API一样，可以发送命令得到所有索引的信息（使用纯_stats端点），或得到某特定索引的信息（例如，library/_stats），或者一次得到几个索引的信息（例如，library,map/_stats）。例如，为了检查本书中使用的map和library索引的统计信息，可以执行如下命令：

```
curl localhost:9200/library,map/_stats?pretty
```

上面的命令将返回超过500行的响应，所以省略它，只描述一下它的结构。除了响应状态和响应时间等信息之外，还可以看到三个对象，分别是primaries、total和indices。indices对象包含library和map索引的信息。primaries对象包含分配在当前节点的主分片信息，total对象包含所有分片（包括副本分片）的所有信息。所有这些对象都包含描述特殊统计的对象，比如docs、store、indexing、get、search、merges、refresh、flush、warmer、filter_cache、id_cache、fielddata、percolate、completion、segments和translog。来讨论存储在这些对象中的信息。

1. docs

响应中的docs节点显示索引文档的信息。它看上去可能如下所示：

```
"docs" : {
 "count" : 4,
 "deleted" : 0
}
```

主要信息是count，指文档的数目。从索引中删除文档时，Elasticsearch并不会立即移除这些文档，只把它们标记为删除。这些文档将在段合并过程中被删除。被标记成删除的文档数目将出现在deleted属性中，在合并结束时应该为0。

2. store

下一个统计组是store，提供了关于存储的信息。例如，该节点看上去可能如下所示：

```
"store" : {
 "size_in_bytes" : 6003,
 "throttle_time_in_millis" : 0
}
```

主要的信息是关于这个索引（或多个索引）的大小。我们还看到了调节统计值。在系统有I/O性能问题，并且设置了段合并过程中内部操作的限制时，这个信息很有用。

3. indexing、get和search

响应中的indexing、get和search节点提供了数据操纵的相关信息：索引删除操作、实时的get和搜索。来看看Elasticsearch返回的下列例子：

```
"indexing" : {
 "index_total" : 11501,
 "index_time_in_millis" : 4574,
 "index_current" : 0,
 "delete_total" : 0,
 "delete_time_in_millis" : 0,
 "delete_current" : 0
},
"get" : {
 "total" : 3,
 "time_in_millis" : 0,
 "exists_total" : 2,
 "exists_time_in_millis" : 0,
 "missing_total" : 1,
 "missing_time_in_millis" : 0,
 "current" : 0
},
"search" : {
 "query_total" : 0,
 "query_time_in_millis" : 0,
 "query_current" : 0,
 "fetch_total" : 0,
 "fetch_time_in_millis" : 0,
 "fetch_current" : 0
}
```

可以看到，所有这些统计具有类似结构。可以看到各种类型的请求花费的总时间（以毫秒为单位）以及请求数，可以用总时间计算单个查询的平均时间。在实时的情况下，`get`请求中有价值的信息是不成功读取的次数（因为文档缺失）。

4. 额外信息

此外，Elasticsearch提供了下列信息。

- `merges`：该节点包含Lucene段合并的信息。
- `refresh`：该节点包含刷新操作的信息。
- `flush`：该节点包含清理信息。
- `warmer`：该节点包含预热器的信息，以及它们执行了多久。
- `filter_cache`：这些是过滤器缓存统计信息。
- `id_cache`：这些是标识符缓存统计信息。
- `fielddata`：这些是字段数据缓存统计信息。
- `percolate`：该节点包含预匹配器使用情况的信息。
- `completion`：该节点包含自动完成建议器的信息。
- `segments`：该节点包含Lucene段的信息。
- `translog`：该节点包含事务日志计数和大小的信息。

8

8.2.3　状态API

另一个得到关于索引信息的方法，是使用_status端点的状态API。返回的信息描述了可用的分片，包含的信息有：哪个分片被认为是主分片，它被分配到了哪个节点，被重分配到了哪个节点（如果是的话），分片的状态（活跃与否），事务日志，合并过程以及刷新和清理统计。

8.2.4　节点信息API

节点信息API提供了关于集群中节点的信息，为从该API得到信息，需要发送请求到_nodes端点。

这个API可以使用如下特性获取单个或特定几个节点的信息。

- Node名：如果想得到名为Pulse的节点的信息，可以在_nodes/Pulse REST端点上执行命令。
- Node标识符：如果想得到标识符为ny4hftjNQtuKMyEvpUdQWg的节点的信息，可以在_nodes/ny4hftjNQtuKMyEvpUdQWg端点上执行命令。
- IP地址：如果想得到IP地址为192.168.1.103的节点的信息，可以在_nodes/192.168.1.103 REST端点上执行命令。
- Elasticsearch配置参数：如果想得到node.rack属性等于2的所有节点的信息，可以在_nodes/rack:2 REST端点上执行命令。

该API还允许使用如下方式一次性得到几个节点的信息：

- 模式，例如，_nodes/192.168.1.*或者_nodes/P*;
- 节点枚举，例如，_nodes/Pulse,Slab;
- 模式与节点枚举，例如，/_nodes/P*,S*。

默认情况下，对节点API的请求将返回关于节点的基本信息，比如名称、标识符、地址。通过添加额外参数，可以获得其他信息。可用的参数如下所示。

- settings：此参数用来获取Elasticsearch配置信息。
- os：此参数用来获取服务器的信息，诸如处理器、内存和交换区等。
- process：此参数用来获取进程标识符和可用的文件描述符等信息。
- jvm：此参数用来获取关于Java虚拟机（JVM）的信息，比如内存限制。
- thread_pool：此参数用来获取各种操作的线程池配置。
- network：此参数用来获取网络接口的名称和地址。
- transport：此参数用来获取传输的侦听接口地址。
- http：此参数用来获取HTTP侦听地址。
- plugins：此参数用来获取安装的插件信息。

使用先前描述的API示例如下：

```
curl 'localhost:9200/_nodes/Pulse/os,jvm,plugins?pretty'
```

上述命令除了返回基本信息外，还有操作系统、Java虚拟机和插件的相关信息。当然，所有信息都是针对Pulse节点。

8.2.5 节点统计API

节点统计API跟前面描述的节点信息API类似。主要的区别是，节点信息API提供环境信息，而节点统计API告诉我们集群工作时发生过什么。为使用节点统计API，你应该发送命令到/_nodes/stats REST端点。跟节点信息API类似，我们同样可以获取特定节点的信息（比如，_nodes/Pulse/stats）。

默认情况下，Elasticsearch返回所有可用的统计，但可以限制为我们感兴趣的那些。可用的选项如下。

- indices：提供了关于索引的信息，包括大小、文档个数、索引相关的统计、搜索和获取时间、缓存、段合并，等等。
- os：提供了操作系统相关的信息，比如可用磁盘空间、内存、交换分区的使用。
- process：提供了跟Elasticsearch进程相关的包括内存、CPU和文件句柄等的使用情况。
- jvm：提供了关于Java虚拟机内存和垃圾回收统计的信息。
- network：提供了TCP级别的信息。
- transport：提供了传输模块发送和接收数据的信息。
- http：提供了HTTP连接的信息。
- fs：提供了可用磁盘空间和I/O操作统计的信息。
- thread_pool：提供了分配给各种操作线程的状态信息。
- breaker：提供了字段数据缓存断路器的信息。

一个使用该API的示例代码如下所示：

```
curl 'localhost:9200/_nodes/Pulse/stats/os,jvm,breaker?pretty'
```

8.2.6 集群状态API

Elasticsearch提供的另一个API是集群状态API。顾名思义，它允许获取关于整个集群的信息（也可以通过在请求中添加local=true，限制只返回本地节点的信息）。用于获取所有信息的基本命令如下所示：

```
curl 'localhost:9200/_cluster/state?pretty'
```

不过，也可以把提供信息限定为特定的度量（用逗号分隔，在REST调用的_cluster/state部分之后指定），以及特定的索引（同样用逗号分隔，在REST调用的_cluster/state/metrics部分之后指定）。下面是一个示例调用，它只返回关于map和library索引的节点相关信息：

```
curl 'localhost:9200/_cluster/state/nodes/map,library?pretty'
```

可使用下面这些度量。

- version：返回关于集群状态版本的信息。
- master_node：返回关于所选主节点的信息。
- nodes：返回节点上的信息。
- routing_table：返回路由相关的信息。
- metadata：返回元数据相关的信息。当指定要获取元数据度量，还可以包含一个额外的参数index_templates=true，将在结果中包含定义的索引模板。
- blocks：返回关于块的信息。

8.2.7　挂起任务API

Elasticsearch 1.0中引入的一个API是挂起任务API（pending tasks API），它允许我们坚持哪些任务在等待执行。为获取这个信息，需要发送请求到/_cluster/pending_tasks REST端点。在响应中，我们将看到一组任务，包含任务优先级、队列等待时间等信息。

8.2.8　索引段API

我们想提的最后一个API是通过使用/_segments端点的Lucene段API。可以为整个集群或单独的索引执行它。该API提供的信息包括分片、分片的布局，以及Lucene库管理的跟物理索引相连的段。

8.2.9　cat API

当然，可以说我们需要的用来诊断和观察集群的所有信息都可以通过提供的API获取。然而，API返回的响应是JSON格式，它很好，但至少对于人类来说，不是特别方便使用。这就是为什么Elasticsearch允许使用一个友好的API：cat API。为使用cat API，需要向_cat REST端点发送一个请求，紧跟着下面的其中一个选项。

- aliases：返回有关别名的信息（8.6节中将介绍别名）。
- allocation：返回分片分配和磁盘使用的信息。
- count：为所有索引或单个索引返回文档个数的信息。
- health：返回集群健康度的信息。

- indices：返回所有索引或单个索引的信息。
- master：返回当选主节点的信息。
- nodes：返回集群拓扑相关信息。
- pending_tasks：返回正在等待执行的任务信息。
- recovery：返回还原过程的视图。
- thread_pool：返回集群范围内的线程池的统计信息。
- shards：返回关于分片的信息。

这可能有点混乱，来看一个返回分片信息的示例命令，如下所示：

```
curl -XGET 'localhost:9200/_cat/shards?v'
```

> 注意，在请求中包括v参数。这意味着我们希望更详细的信息，例如，包括头部信息。除v参数外，也可以使用help参数，它将返回给定命令的头部描述；还有h参数，它接受一个逗号分隔的列表，指定要包含在响应中的列。

上述命令的响应如下所示：

```
index     shard  prirep  state    docs  store   ip            node
map       0      p       STARTED  4     5.9kb   192.168.1.40  es_node_1
library   0      p       STARTED  9     11.8kb  192.168.56.1  es_node_2
```

可以看到，我们有两个索引，每个都有一个分片。还看到分片的ID，也就是说，它是不是主分片，以及它的状态、文档数、大小、节点的IP地址、节点名称等。

限制返回信息

一些cat API命令允许限制它们返回的信息。例如，别名调用允许我们通过添加别名来得到特定别名的信息，就像下面的命令：

```
curl -XGET 'localhost:9200/_cat/aliases/current_index'
```

我们总结一下允许限制信息的命令。

- aliases：通过在请求中添加别名，来限定获取特定别名的信息。
- count：通过在请求中添加我们感兴趣的索引名字，来限定获取特定索引的信息。
- indices：同count一样，限定只获取特定索引的信息。
- shards：通过在请求中添加我们感兴趣的索引名字，来限定获取特定索引的信息。

8.3　控制集群的再平衡

默认情况下，Elasticsearch试图把分片和其副本在集群中均衡分布。在大多数情况下这种行为是好的，但有时我们想控制此行为。本节将深入介绍如何避免集群再平衡（rebalancing），以及如何控制这一过程的行为。

假设你的网络可以处理很高的流量，或者相反，网络被大量使用，你希望可以避免太多压力。另一个例子是，你可能想在整个集群重启后，减少I/O子系统的压力，并且同时你要减少分片和副本的初始化。这只是两个再平衡控制可能很方便的例子。

8.3.1　再平衡

再平衡是在集群的不同节点之间移动分片的过程。我们已经提到，在大多数情况下它是好的，但有时你可能想完全避免这种情况。如果我们定义且希望保持分片放置，就要避免再平衡。然而，默认情况下，当集群状态变化，或者Elasticsearch认为需要再平衡时，它会尽量对集群进行再平衡。

8.3.2　集群的就绪

我们已经知道，索引可以由分片和副本构成。主分片（或者简称分片）用于新文档被编入索引以及更新或删除，或者只是索引发生任何变化时。我们的副本从主分片获取数据。

你可以认为，当所有主分片都被分配在集群中的节点上，也就是一旦达到黄色的健康状态时，集群就已经就绪了。然而，此时Elasticsearch还可能在初始化其他分片：副本。但是，你已经可以使用集群，并确保可以搜索整个数据集，也可以发送索引更改命令。之后，这些都将被正确处理。

8.3.3　集群再平衡设置

Elasticsearch允许控制再平衡过程，通过设置elasticsearch.yml文件中的几个属性，或使用Elasticsearch REST API（在8.8节中描述）。

1. 控制再平衡何时开始

可以通过设置`cluster.routing.allocation.allow_rebalance`属性来指定再平衡何时开始。该属性可以设置为如下几个值。

- `always`：该值表明再平衡可以在需要时随时开始。
- `indices_primaries_active`：该值表明当所有的主分片都初始化后，再平衡才会开始。

- indices_all_active：该值是默认设置，意味着所有分片和副本都初始化后，再平衡才会开始。

2. 控制同时在节点中移动的分片数量

可以设置cluster.routing.allocation.cluster_concurrent_rebalance属性来指定整个集群中同时可以在节点间移动的分片数量。如果你的集群由很多节点组成，可以提高这个值，默认值为2。

3. 控制单个节点上同时初始化的分片数量

cluster.routing.allocation.node_concurrent_recoveries属性可以用来设置Elasticsearch在单个节点上一次可以初始化多少分片。请注意分片的还原过程是非常耗I/O的，所以你可能要避免太多的分片同时被还原。该属性值默认跟上一个属性一样，为2。

4. 控制单个节点上同时初始化的主分片数

可以设置cluster.routing.allocation.node_initial_primaries_recoveries属性来控制单个节点上一次可以初始化多少主分片。

5. 控制分配的分片类型

使用cluster.routing.allocation.enable属性，可以控制允许分配哪种类型的分片。该属性可以使用如下值。

- all：这是默认值，告诉Elasticsearch所有类型的分片都可以被分配。
- primaries：告诉Elasticsearch它应该只分配主分片，而不要分配副本。
- new_primaries：告诉Elasticsearch只分配新创建的主分片。
- none：这完全禁用了分片的分配。

6. 控制单个节点上的并发流数目

indices.recovery.concurrent_streams属性允许控制在一个节点上一次可以打开多少流，以便从目标分片中恢复一个分片。它的默认值为3。如果你的网络和节点可以处理更多，就可以提高这个值。

8.4　控制分片和副本的分配

Elasticsearch集群内部的索引，可以由许多分片建成，每个分片可以有多个副本。因为单一的索引可以有多个分片，可以处理索引大到无法存入到单台机器的情况。可能有其他原因，比如与存储相关的内存和CPU等。因为每个分片可以有多个副本，可以通过把副本散布到多台服务器

8

上，来处理更高的查询。可以说通过使用分片和副本，可以横向扩展Elasticsearch。然而，Elasticsearch必须找出在集群的什么地方放置分片和副本，即每个分片或副本应放在哪些服务器或节点上。

8.4.1　显式控制分配

假设希望把索引放置在不同的集群节点上。例如，把一个叫shop的索引放置在某些节点上，第二个叫users的索引放置在其他节点上。把最后一个名为promotions的索引放置在users和shop索引放置的所有节点上。需要这样做可能是由于性能原因。我们知道安装过Elasticsearch的一些服务器比其他服务器更强大。使用默认Elasticsearch行为，我们不知道分片和副本将放置在哪，但幸好，Elasticsearch允许我们控制它。

1. 指定节点参数

所以让我们把集群分为两个区域。我们说“区”，但它可以是任何你喜欢的名称，我们只是喜欢用“区”。假设有四个节点，希望把更强大的编号为1和2的节点放置在一个叫zone_one的区；编号3和4的节点资源较少，放在叫zone_two的区域。

2. 配置

为了实现我们描述的索引分布，在节点1和节点2（较强的节点）的elasticsearch.yml配置文件中添加node.zone: zone_one属性。在节点3和节点4（较弱的节点）的elasticsearch.yml配置文件中添加类似的属性：node.zone: zone_two。

3. 索引的创建

现在创建索引。首先创建shop索引。将此索引放置到更强的节点。为此可以运行以下命令：

```
curl -XPUT 'http://localhost:9200/shop' -d '{
 "settings" : {
  "index" : {
   "routing.allocation.include.zone" : "zone_one"
  }
 }
}'
```

上述命令将创建shop索引并指定index.routing.allocation.include.zone属性的值为zone_one，这意味着我们希望把shop索引放到node.zone属性等于zone_one的节点上。

为users索引执行类似的步骤：

```
curl -XPUT 'http://localhost:9200/users' -d '{
 "settings" : {
  "index" : {
   "routing.allocation.include.zone" : "zone_two"
```

```
    }
  }
}'
```

不过这一次，我们指定希望把users索引放到node.zone属性等于zone_two的节点上。

最后，promotions索引应该放到上述所有节点，因此使用下列命令来创建和配置这个索引：

```
curl -XPOST 'http://localhost:9200/promotions'
curl -XPUT 'http://localhost:9200/promotions/_settings' -d '{
 "index.routing.allocation.include.zone" : "zone_one,zone_two"
}'
```

这次使用了不同的命令集。第一个命令创建索引，第二个命令更新index.routing.allocation.include.zone属性的值。这样做是为了说明可以用这种方式来做。

4. 排除节点的分配

既然可以指定索引应该放在哪个节点，就可以指定应该排除哪些节点。参考之前的例子，如果希望名为pictures的索引不放在node.zone属性等于zone_one的节点上，可以执行以下命令：

```
curl -XPUT 'localhost:9200/pictures/_settings' -d '{
 "index.routing.allocation.exclude.zone" : "zone_one"
}'
```

注意，这次使用index.routing.allocation.exclude.zone属性，而不是index.routing.allocation.include.zone属性。

5. 节点需求属性

除了节点的包含和排除规则，还可以指定分片必须匹配某种规则才能分配到给定节点上。不同的是，使用index.routing.allocation.include属性时，索引将被放置到与该属性中至少一个值匹配的节点上。而使用index.routing.allocation.require属性值时，Elasticsearch将把索引放置到与该属性的所有值都匹配的节点上。例如，我们已经为pictures索引做了如下设置：

```
url -XPUT 'localhost:9200/pictures/_settings' -d '{
 "index.routing.allocation.require.size" : "big_node",
 "index.routing.allocation.require.zone" : "zone_one"
}'
```

执行以上命令后，Elasticsearch将只会把pictures索引的分片分配到node.size属性等于big_node且node.zone属性等于zone_one的节点上。

6. 使用IP地址分配分片

除了在节点的配置中添加一个特殊的参数，也可以使用IP地址来指定应该包含或排除哪些节点用来做分片和副本的分配。为此，替换index.routing.allocation.include.zone或index.routing.allocation.exclude.zone属性的zone部分，而改用_ip。如果希望把shop

索引只放置到IP地址为10.1.2.10和10.1.2.11的节点上，执行以下命令：

```
curl -XPUT 'localhost:9200/shop/_settings' -d '{
 "index.routing.allocation.include._ip" : "10.1.2.10,10.1.2.11"
}'
```

7. 基于磁盘的分片分配

除了上面描述的分片过滤方法外，Elasticsearch 1.0引入了一个额外的基于磁盘的方法，它允许基于节点的磁盘使用情况来设置分配规则，因此不会有耗尽磁盘空间或类似的问题。

(1) 启用基于磁盘的分片分配

基于磁盘的分片分配默认是禁用的。可以通过设置`cluster.routing.allocation.disk.threshold_enabled`属性为`true`来启用它。可以在elasticsearch.yml文件中设置，或者使用集群设置API动态设置（8.8节会介绍）：

```
curl -XPUT localhost:9200/_cluster/settings -d '{
 "transient" : {
  "cluster.routing.allocation.disk.threshold_enabled" : true
 }
}'
```

(2) 配置基于磁盘的分片分配

以下三个属性可以用来控制基于磁盘的分片分配的行为，它们都可以动态更新，或者在elasticsearch.yml配置文件中设置。

第一个属性是`cluster.info.update.interval`，默认值为30秒，定义了Elasticsearch更新节点上磁盘使用信息的时间间隔。

第二个属性是`cluster.routing.allocation.disk.watermark.low`，默认值为0.70。这意味着Elasticsearch不会在磁盘空间被使用超过70%的节点上分配新的分片。

第三个属性是`cluster.routing.allocation.disk.watermark.high`，默认值为0.85。意味着Elasticsearch对磁盘使用大于等于85%的节点将开始重新分配分片。

`cluster.routing.allocation.disk.watermark.low`和`cluster.routing.allocation.disk.watermark.high`属性都可以设置成一个百分比（比如0.60，表示60%），或者一个绝对值（600 mb，表示600兆字节）

8.4.2 集群范围的分配

除了在索引级别上指定分配的包含和排除规则（到目前为止我们讨论的），还可以在集群中的所有索引上指定。如果希望把所有新索引都放置到IP地址为10.1.2.10和10.1.2.11的节点上，执

行如下命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
 "transient" : {
  "cluster.routing.allocation.include._ip" : "10.1.2.10,10.1.2.11"
 }
}'
```

我们注意到，命令发送到_cluster/settings REST端点，而不是INDEX_NAME/_settings端点。当然，可以使用在索引级别上的所有包含、排除、需求规则。

请注意，集群的瞬时和永久属性在8.8节讨论。

8.4.3　每个节点上的分片和副本数量

除了指定分片和副本的分配外，还可以指定单一节点上为单一索引最多可以放置多少分片。如果希望shop索引在每个节点上只有一个分片，执行以下命令：

```
curl -XPUT 'localhost:9200/shop/_settings' -d '{
 "index.routing.allocation.total_shards_per_node" : 1
}'
```

该属性可以放在elasticsearch.yml文件中，或者使用上述命令更新生产环境的索引。请记住，如果Elasticsearch无法分配所有的主分片，你的集群将维持在红色状态。

8.4.4　手动移动分片和副本

我们想讨论的最后一件事是手动在节点间移动分片的能力。这可能有用，例如，在关掉一个节点之前，你想把该节点上的所有分片移动到其他地方。Elasticsearch公开了_cluster/reroute REST端点，允许我们控制这个。

有以下可用的操作：

- 把分片从一个节点移到另一个节点；
- 取消分片的分配；
- 强制分片的分配。

现在让我们仔细看看上述操作。

1. 移动分片

假设有两个节点，分别为es_node_one和es_node_two。除此之外，我们的shop索引有两个分片被Elasticsearch放置在第一个节点上。现在，我们想把第二个分片移动到第二个节点上。为此，可以执行以下命令：

8

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
 "commands" : [ {
  "move" : {
   "index" : "shop",
   "shard" : 1,
   "from_node" : "es_node_one",
   "to_node" : "es_node_two"
  }
 } ]
}'
```

我们指定了move命令，它允许移动由index属性指定的索引的分片（和副本）。shard属性是要移动的分片的编号。最后，from_node属性指定了希望从哪个节点上移动分片，to_node属性指定了希望把分片放置到哪个节点上。

2. 取消分片的分配

如果想取消正在进行的分配过程，可以执行cancel命令，并指定想取消分片的索引、节点和分片。例如以下命令：

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [ {
    "cancel" : {
    "index" : "shop",
    "shard" : 0,
    "node" : "es_node_one"
    }
  } ]
}'
```

上述命令将取消es_node_one节点上shop索引编号为0的分片分配。

3. 强制分片的分配

除了取消和移动分片和副本，还可以分配一个未分配的分片到指定节点上。如果我们的users索引有个编号为0的未分配分片，想把它分配到es_node_two节点上，执行以下命令：

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [ {
    "allocate" : {
    "index" : "users",
    "shard" : 0,
    "node" : "es_node_two"
   }
  } ]
}'
```

4. 每个HTTP请求多个命令

当然，也可以在单个HTTP请求中包含多个命令，例如，考虑以下命令：

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
 "commands" : [
  {"move" : {"index" : "shop", "shard" : 1, "from_node" : "es_node_one",
"to_node" : "es_node_two"}},
  {"cancel" : {"index" : "shop", "shard" : 0, "node" : "es_node_one"}}
 ]
}'
```

8.5 预热

有时，可能需要为了处理查询而准备Elasticsearch。也可能因为你严重依赖字段数据缓存，需要在生产查询到达之前加载它们，或者你可能想预热操作系统I/O缓存。不管什么原因，Elasticsearch允许为类型和索引定义预热查询（warning query）。

8.5.1 定义一个新的预热查询

预热查询跟普通查询没什么区别，只是它存储在Elasticsearch一个特殊的名为_warmer的索引中。假设想使用下面的查询来做预热：

```
{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "warming_facet" : {
      "terms" : {
        "field" : "tags"
      }
    }
  }
}
```

为了把上述查询存储为library索引的预热查询，执行以下命令：

```
curl -XPUT 'localhost:9200/library/_warmer/tags_warming_query' -d '{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "warming_facet" : {
      "terms" : {
        "field" : "tags"
      }
    }
  }
}'
```

上诉命令将我们的查询注册为一个名为tags_warming_query的预热查询。你的索引可以

有多个预热查询，但每一个查询都需要一个唯一的名称。

我们不仅可以对整个索引定义预热查询，还可以为索引中的特定类型定义。为了把前面展现的查询存为library索引中book类型的预热查询，执行之前的命令，但不再是/library/_warmer URI，而是/library/book/_warmer。所以，整个命令将如下所示：

```
curl -XPUT 'localhost:9200/library/book/_warmer/tags_warming_query' -d '{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "warming_facet" : {
      "terms" : {
        "field" : "tags"
      }
    }
  }
}'
```

添加一个预热查询后，Elasticsearch允许一个新段执行搜索之前，会在那个段上执行定义的预热查询。它允许Elasticsearch和操作系统缓存数据，以此来加速搜索。

> 正如1.1节中介绍的，Lucene把索引分为多个段，段一旦写入不再修改。每个新的提交操作都会创建一个新段（如果段的数量太大，将会被合并），Lucene使用它来搜索。

8.5.2　获取定义的预热查询

为了获取索引中的特定预热查询，我们只需知道它的名称。如果想获取library索引中名为tags_warming_query的预热查询，执行下面的命令：

```
curl -XGET 'localhost:9200/library/_warmer/tags_warming_query?pretty=true'
```

Elasticsearch返回的结果将如下所示（注意我们使用了pretty=true参数让响应更易阅读）：

```
{
  "library" : {
    "warmers" : {
      "tags_warming_query" : {
        "types" : [ ],
        "source" : {
          "query" : {
            "match_all" : { }
          },
          "facets" : {
            "warming_facet" : {
              "terms" : {
                "field" : "tags"
```

```
                    }
                  }
                }
              }
            }
          }
        }
      }
```

也可以使用下面的命令来获取索引和类型的所有预热查询：

```
curl -XGET 'localhost:9200/library/_warmer'
```

最后，还可以获取所有以特定前缀开头的预热查询。例如，想得到library索引上所有以tags前缀开头的预热查询，执行下面的命令：

```
curl -XGET 'localhost:9200/library/_warmer/tags*'
```

8.5.3 删除一个预热查询

删除预热查询跟获取预热查询非常类似，只需使用DELETE这个HTTP方法。为了从索引中删除一个特定的预热查询，我们只需知道它的名称。例如，从library索引中删除名为tags_warming_query的预热查询，执行以下命令：

```
curl -XDELETE 'localhost:9200/library/_warmer/tags_warming_query'
```

也可以使用下面命令来删除索引上的所有预热查询：

```
curl -XDELETE 'localhost:9200/library/_warmer/_all'
```

最后，也可以删除所有以给定前缀开头的预热查询。例如，要删除library索引上的所有以tags前缀开头的预热查询，可执行以下命令：

```
curl -XDELETE 'localhost:9200/library/_warmer/tags*'
```

8.5.4 禁用预热功能

为了完全禁用预热查询，但仍把它们保存在_warmer索引中，你应该设置index.warmer.enabled属性为false（设置为true则启用预热功能）。该属性可以在elasticsearch.yml文件中设置，也可以在生产集群上使用REST API。

如果想禁用library索引上的预热功能，执行以下命令：

```
curl -XPUT 'http://localhost:9200/library/_settings' -d '{
  "index.warmer.enabled" : false
}'
```

8.5.5　查询的选择

你可能会问哪个查询应该用做预热查询。通常要选择执行起来昂贵的和需要填充缓存的查询。因此，可能会选择基于索引中的字段做切面和排序的查询。除此以外，父子查询和包括常用过滤器的查询也可考虑。你也可以通过查看日志选择其他查询，找到性能表现不尽人意的，这些也可以是很好的预热候选对象。

假设在elasticsearch.yml文件中设置了以下的日志配置：

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 1s
```

并且在logging.yml配置文件中设置了以下日志级别：

```
logger:
  index.search.slowlog: TRACE, index_search_slow_log_file
```

注意，`index.search.slowlog.threshold.query.trace`属性设置成1s，而日志级别`index.search.slowlog`属性设置成`TRACE`。这意味着每当一个查询执行时间查过1秒（在分片上，不是总时间）时，它将被记录到慢查询日志文件中（日志文件由logging.yml配置文件中的`index_search_slow_log_file`节点指定）。例如，慢查询日志文件中可能找到下面的条目：

```
[2013-01-24 13:33:05,518][TRACE][index.search.slowlog.query] [Local
test] [library][1] took[1400.7ms], took_millis[1400], search_
type[QUERY_THEN_FETCH], total_shards[32], source[{"query":{"match_
all":{}}}], extra_source[]
```

可以看到，前面的日志行包含查询时间、搜索类型和搜索源本身，它显示了执行的查询。

当然，你的配置文件中可能有不同的值，但慢查询日志可以是一个很有价值的源，从中可以找到执行时间过长的、可能需要定义预热的查询，它们可能是父子查询，需要获取一些标识符来提高性能，或者你第一次使用了一些费时的过滤器？

> 应该记住，不要让你的Elasticsearch集群加载过多的预热查询，因为最终你可能会花太多的时间预热，而不是处理你的生产查询。

8.6　使用索引别名来简化你的日常工作

当在Elasticsearch中使用多个索引时，你可能有时会忘记它们。想象你在索引中存储日志的场景。通常，日志消息的数量相当大，因此，把数据划分是一个好的解决方案。一种逻辑划分方法是为每天的日志创建一个索引（如果你对管理日志的开源方案感兴趣，可以看看

http://logstash.net上的Logstash）。但一段时间后，如果保存了所有索引，就会开始头疼如何管理它们。应用程序需要管理所有这些信息，比如发送数据到哪个索引，查询哪个索引，等等。通过使用别名，可以改变这个状况，使用一个名字来跟多个索引打交道，就像使用一个索引一样。

8.6.1 别名

什么是索引别名？它是一个或多个索引的一个附加名称，允许使用这个名称来查询索引。一个别名可以对应多个索引，反之亦然，一个索引可以是多个别名的一部分。然而，请记住，你不能使用对应多个索引的别名来进行索引或实时的GET操作。如果你这样做，Elasticsearch将抛出一个异常（但可以使用对应单个索引的别名来进行索引操作），因为Elasticsearch不知道应该把索引建立到哪个索引上，或从哪个索引获取文档。

8.6.2 创建别名

为创建一个索引别名，需要在_aliases REST端点上执行一个HTTP POST方法。例如，如下的请求将创建一个名为week12的别名，它包含day10、day11和day12这些索引。

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "add" : { "index" : "day10", "alias" : "week12" } },
    { "add" : { "index" : "day11", "alias" : "week12" } },
    { "add" : { "index" : "day12", "alias" : "week12" } }
  ]
}'
```

如果我们的Elasticsearch集群中不存在week12别名，上述命令将创建它。如果存在，该命令将只是把指定的索引添加进去。

之前，执行一个跨三个索引的搜索是这样的：

```
curl -XGET 'localhost:9200/day10,day11,day12/_search?q=test'
```

如果一切顺利，现在可以这样执行：

```
curl -XGET 'localhost:9200/week12/_search?q=test'
```

这不是更好吗？

8.6.3 修改别名

当然，你可以从别名中移除索引。这与添加索引到别名类似，但使用remove命令，而不是add。例如，为了从week12中移除day9索引，执行以下命令：

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "remove" : { "index" : "day9", "alias" : "week12" } }
  ]
}'
```

8.6.4　合并命令

add和remove命令可以在一个请求中发送。例如，想合并之前发送的所有命令到一个单一的请求中，可以发送以下命令：

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "add" : { "index" : "day10", "alias" : "week12" } },
    { "add" : { "index" : "day11", "alias" : "week12" } },
    { "add" : { "index" : "day12", "alias" : "week12" } },
    { "remove" : { "index" : "day9", "alias" : "week12" } }
  ]
}'
```

8.6.5　获取所有别名

除了对别名添加或移除索引，使用Elasticsearch的应用程序，可能需要获取集群中的所有别名或跟一个索引连接的所有别名。为了获取这些别名，发送一个HTTP GET命令。例如，下面的第一个命令获取day10索引的所有别名，第二个命令获取所有可能的别名：

```
curl -XGET 'localhost:9200/day10/_aliases'
curl -XGET 'localhost:9200/_aliases'
```

第二个命令返回的响应如下：

```
{
  "day10" : {
    "aliases" : {
      "week12" : { }
    }
  },
  "day11" : {
    "aliases" : {
      "week12" : { }
    }
  },
  "day12" : {
    "aliases" : {
      "week12" : { }
    }
  }
}
```

8.6.6 移除别名

可以使用_alias端点移除一个别名。例如，发送下面命令将从data索引移除client别名：

```
curl -XDELETE localhost:9200/data/_alias/client
```

8.6.7 别名中的过滤

可以像在SQL数据库中使用视图一样来使用别名。你可以使用完整的查询DSL（第2章详细讨论过），将你的查询应用到所有的count、search、delete，等等。

来看一个例子。假设想要一个别名返回特定客户的数据，以便在应用程序中使用。客户标识符存在clientId字段上，我们对客户12345感兴趣。所以，在数据索引中创建一个名为client的别名，它自动在clientId上应用一个过滤器：

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
  {
    "add" : {
      "index" : "data",
      "alias" : "client",
      "filter" : { "term" : { "clientId" : "12345" } }
    }
  } ]
}'
```

所以，当使用该别名时，你的请求将总是被一个词条查询过滤，确保所有文档的clientId字段都等于12345。

8.6.8 别名和路由

跟在别名中使用过滤器类似，可以在别名中添加路由值。假设我们在使用基于用户标识符的路由，且想在别名中使用相同的路由值，则在名为client的别名中，我们将在查询中使用路由值12345、12346、12347，而在索引建立中只使用12345。为此，使用下面的命令创建别名：

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
  {
    "add" : {
      "index" : "data",
      "alias" : "client",
      "search_routing" : "12345,12346,12347",
      "index_routing" : "12345"
    }
  } ]
}'
```

这样，使用client别名索引数据时，将使用index_routing属性的值。在查询时，将使用search_routing属性值。

还有一件事。请看发送到上述别名的如下查询：

```
curl -XGET 'localhost:9200/client/_search?q=test&routing=99999,12345'
```

它将使用12345作为路由值。这是因为Elasticsearch将使用search_routing属性值和查询路由参数值的共同值，在我们的例子中是12345。

8.7　Elasticsearch 插件

在本书不同的地方，我们使用了不同的Elasticsearch插件。你可能还记得6.5节描述过的，脚本中使用的额外编程语言和对附件的支持。本节将介绍插件如何工作以及安装。

8.7.1　基础知识

Elasticsearch插件位于plugins目录的各自子目录中。如果从网站上下载一个新的插件，你可以用该插件名创建一个新目录，并把插件存档解压到这个目录中。还有个更方便的安装方法：使用插件脚本。我们在本书中已经用过几次，所以现在该描述这个工具了。

Elasticsearch有两种主要类型的插件。这两种类型可以基于其内容来分类：Java插件和站点插件（site plugins）。Elasticsearch把站点插件当成被内置的HTTP服务器处理的文件集，处于/_plugin/plugin_name/URL（比如/_plugin/bigdesk）下面。此外，任何一个没有Java内容的插件将被自动视为站点插件，就这么简单。在Elasticsearch看来，站点插件不会改变Elasticsearch的行为。Java插件通常包含.jar文件，被es-plugin.properties文件扫描。该文件包含主要类的信息，Elasticsearch使用该类作为入口来配置插件并扩展Elasticsearch的功能。Java插件可以包含站点部分，被内置的HTTP服务器使用（跟站点插件一样），这部分需要放置在_site目录中。

8.7.2　安装插件

默认情况下，插件从网站下载。如果该网站找不到插件，将检查Maven Central（http://search.maven.org/）、Maven Sonatype(https://repository.sonatype.org/)和GitHub（https://github. com/)等库。插件工具假定给定的插件地址由组织名称、插件名称和版本号组成。看看下面的命令：

```
bin/plugin -install elasticsearch/elasticsearch-lang-javascript/2.0.0.RC1
```

上述命令将安装一个插件，该插件允许使用额外的脚本语言：JavaScript。选择该插件的2.0.0.RC1版本。也可以省略版本号，那样，Elasticsearch将尝试寻找跟Elasticsearch版本号一样的版本或者最新的主版本。

我们期待命令的结果，下面是执行上述命令的一个示例结果：

```
-> Installing elasticsearch/elasticsearch-lang-javascript/2.0.0.RC1...
Trying http://download.elasticsearch.org/elasticsearch/elasticsearch-
lang-javascript/elasticsearch-lang-javascript-2.0.0.RC1.zip...
Downloading .............................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
........................................DONE
Installed elasticsearch/elasticsearch-lang-javascript/2.0.0.RC1 into /
opt/elasticsearch-1.0.0/plugins/lang-javascript
```

如果你编写自己的插件，没有访问上述网站的权限，也没有问题。插件工具提供了`-url`选项，允许为插件设置包括本地文件系统（使用`file://`前缀）在内的任何地址。例如，下面的命令将安装存档在本地文件系统的`/tmp/elasticsearch-lang-javascript-2.0.0.RC1.zip`里的插件：

```
bin/plugin -install lang-javascript -url file:///tmp/elasticsearch-lang-
javascript-2.0.0.RC1.zip
```

8.7.3 移除插件

移除插件跟移除目录一样简单。你也可以使用插件工具来做。例如，为了移除`river-mongodb`插件，可以执行下面的命令：

```
bin/plugin -remove river-mongodb
```

> 你需要重启Elasticsearch节点，以让插件的安装或移除生效。

8.8 更新设置 API

Elasticsearch允许在elasticsearch.yml文件中指定各种参数来调优。但你应该把这个文件当做默认设置，可以在运行时通过Elasticsearch REST API修改。

为了设置其中一个属性，需要使用HTTP PUT方法，发送一个合适的请求到_cluster/setting URI。我们有两个选择：瞬时和永久的属性设置。

第一个，瞬时，将只设置属性直到第一次重启。为此，发送下面的命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "PROPERTY_NAME" : "PROPERTY_VALUE"
  }
}'
```

可以看到，在上面的命令中，我们使用了名为transient的对象，并在其中添加属性定义。这意味着该属性值将生效，直到重新启动。如果希望属性设置在重启之后永久生效，使用名称persistent，而不是transient。

任何时候，都可以使用下列命令来获取这些设置：

```
curl -XGET localhost:9200/_cluster/settings
```

8.9　小结

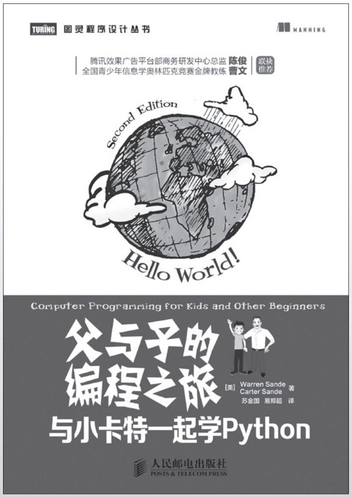
本章介绍了如何为集群创建备份；创建一个备份存储库、创建了备份，并管理它们。我们了解到如何使用Elasticsearch API监控集群，什么是cat API以及为什么它更方便使用。我们还控制了分片的分配，学会了如何在集群中移动分片和控制集群再平衡。我们使用预热功能为生产查询准备集群，学到别名如何帮助我们管理集群中的数据。最后，本章研究了Elasticsearch插件，以及如何使用Elasticsearch提供的更新设置API。

本书内容到此结束了。我们希望你有一个很好的阅读体验并觉得这是本有趣的书。我们真的希望你能有所收获，并从此发现在日常工作中使用Elasticsearch更容易了。作为本书的作者和Elasticsearch用户，我们试图带给读者最好的阅读体验。当然，Elasticsearch还有很多书中没有描述的内容，特别是涉及监控、管理功能和API的内容。图书的页面毕竟是有限的，如果把一切都描述得很详细，这本书就得一千多页了。此外，我们恐怕也无法把一切都描述得足够详细。Elasticsearch不仅是用户友好的，还提供了大量配置选项、查询的可能性等，因此，我们必须选择哪些功能要详细说明，哪些只是一带而过，哪些要完全跳过。希望本书对主题的选择是对的。

还要说的是，记住Elasticsearch在不断演化。在写这本书期间，我们经历了一些稳定的版本，最终发布了1.0.0和1.0.1。即便在这之前，我们也知道会有新功能和改进。如果你想跟进正在添加的新功能，请一定定期检查上新版本的发布说明。我们也会把自认为值得一提的新功能写在上，所以如果你感兴趣，请时常访问此网站。

TURING 图灵教育

图灵最新重点图书

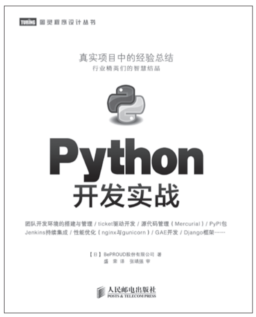


- 程序员爸爸的第一本亲子互动编程书
- 腾讯效果广告平台部商务研发中心总监陈俊
  全国青少年信息学奥林匹克竞赛金牌教练曹文
  联袂推荐
- 内容经过教育专家的评审，经过孩子的亲身检验，并得到了家长的认可

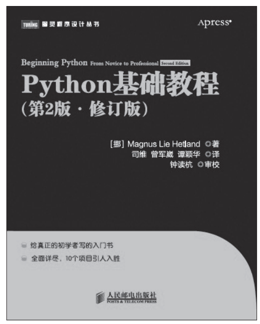
父与子的编程之旅
书号：978-7-115-36717-4
作者：Warren Sande Carter Sande
定价：69.00 元



Python 计算机视觉编程
书号：978-7-115-35232-3
作者：Jan Erik Solem
定价：69.00 元



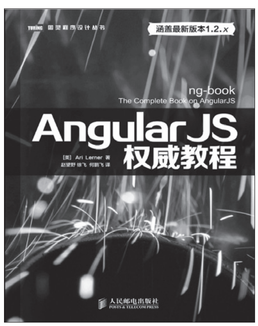
Python 开发实战
书号：978-7-115-32089-6
作者：BePROUD 股份有限公司
定价：79.00 元



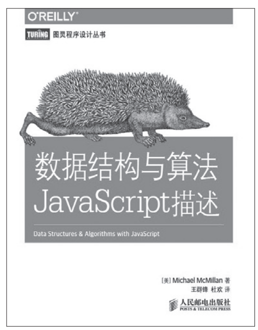
Python 基础教程（第 2 版·修订版）
书号：978-7-115-35352-8
作者：Magnus Lie Hetland
定价：79.00 元



程序员必读之软件架构
书号：978-7-115-37107-2
作者：Simon Brown
定价：49.00 元



AngularJS 权威教程
书号：978-7-115-36647-4
作者：Ari Lerner
定价：99.00 元



数据结构与算法 JavaScript 描述
书号：978-7-115-36339-8
作者：Michael McMillan
定价：49.00 元

Elasticsearch
服务器开发（第2版）

Elasticsearch是基于Lucene的新一代分布式、RESTful风格的开源搜索引擎，具有实时搜索、稳定、快速、安装使用方便等优点。Elasticsearch在全球拥有众多知名用户：GitHub使用Elasticsearch搜索20 TB的数据——包括13亿文件和1300亿行代码，有“音频分享界YouTube”之称的SoundCloud使用Elasticsearch为1.8亿会员在线即时提供音频搜索结果，德国商务社交网站Xing使用Elasticsearch为1400万会员提供可扩展、实时的搜索，手机服务网站Foursquare使用Elasticsearch实时搜索5000万个地点，浏览器插件StumbleUpon利用Elasticsearch每天向它们的社区发送数百万条推荐。

本书介绍了Elasticsearch这个优秀的全文检索和分析引擎从安装和配置到集群管理的各方面知识。本书这一版不仅补充了上一版中遗漏的重要内容，并且所有示例和功能均基于Elasticsearch服务器1.0版进行了更新。你可以从头开始循序渐进地学习本书，也可以查阅具体功能解决手头问题。



[PACKT] PUBLISHING

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/数据库/Elasticsearch

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38032-6

ISBN 978-7-115-38032-6

定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks